ARCHITECTURAL SUPPORT FOR IMPROVING SECURITY AND PERFORMANCE OF MEMORY SUB-SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Chenyu Yan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2008

ARCHITECTURAL SUPPORT FOR IMPROVING SECURITY AND PERFORMANCE OF MEMORY SUB-SYSTEMS

Approved by:

Milos Prvulovic, Advisor
College of Computing
Georgia Institute of Technology

Hyesoon Kim
College of Computing
Georgia Institute of Technology

Gabriel Loh
College of Computing
Georgia Institute of Technology

Yan Solihin
Department of Electrical and Computer Engineering
North Carolina State University

Umakishore Ramachandran
College of Computing
Georgia Institute of Technology

Date Approved: November 14, 2008

To my dear family:

Thank you for all of your love, support and encouragements.

ACKNOWLEDGEMENTS

During my Ph.D. study, I have been extremely lucky to have support, encouragement, and inspiration from many people. Without them, this work would not have been possible.

My sincere and deep gratitude goes to my advisor, Dr. Milos Prvulovic, for his kind guidance and consistent support. He led me on this journey, guided me towards the right direction, influenced me as an active thinker, and provided me with insightful and inspiring advice throughout my entire Ph.D. study. He has been patient and kind, and has always enlightened me with his sparks of wisdom. It has truly been a privilege to work with him.

This work would not have been possible without the support of Brian Rogers and Dr. Yan Solihin. Brian has been a great collaborator and I have benefited greatly from discussions we have had and his knowledge of secure systems. I would also like to thank Dr. Solihin for his tremendous help in guiding the development of many ideas presented in the thesis.

I would also like to thank other members of my committee, Dr. Gabriel Loh, Dr. Umakishore Ramachandran, and Dr. Hyesoon Kim, for their interest in my work. Their insightful comments have significantly improved the quality of my work. I have been very fortunate to be a member of a great computer architecture research group at Georgia Tech. I would also like to thank Dr. Hsien-Hsin Sean Lee for providing many insightful comments.

I also thank Guru Prasadh Venkataramani, Ioannis Doudalis, Samantika Subramaniam, Kiran Puttaswamy, and many other members of archbeer, for their valuable assistance on my research and for their contribution to the wonderful and enjoyable graduate experience I have had at Georgia Tech. They have enriched my life at Georgia Tech and made it happy and memorable. Their friendship has been my best fortune.

I would particularly like to thank my family. Always supportive and encouraging, my parents are my foundation and help take care of my daughter when I was working on my research. Finally, there are no words to express my love to my wife, Ge. Her exceptional

kindness and love makes everything worthwhile.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Computer users are becoming increasingly aware of and concerned for the security of their computer systems. While many security software packages are available, they are *ineffective* in protecting against an emerging class of security attacks that involve directly tampering with the physical operation of the system. Such *hardware attacks* have been demonstrated to be feasible and relatively easy to perform in breaking the Digital Rights Management features in game consoles, and in breaking the security features of a version of a secure processor DS5002FP. These hardware attacks may involve inserting a device on the communication path between the microprocessor and other chips to *passively* snoop their communication, or to *actively* modify data communication between the processor and other chips.

Various schemes were proposed in prior work to provide architectural support for data security and to protect against such hardware attacks. However, such protection schemes usually come with high performance overhead and prohibit the wide adoption of the secure systems. In particular, cryptographic latency was added directly to the memory access latency and introduced significant degradation of the system performance for memory intensive applications. In more recent work, schemes were proposed to reduce the performance overhead through parallelizing the cryptographic operations with memory accesses. However, such schemes usually require a significant on-chip storage area for storing additional security related data. Although these schemes realized performance improvement for secure architectural support, the additional cost of on-chip storage area still is a big obstacle to the wide adoption of the secure systems. We believe secure architectural support can be provided efficiently and effectively without either significant performance degradation or noticeable increase in on-chip area cost.

This thesis explores architectural level optimizations to make secure systems more efficient, secure and affordable. It extends prior work for secure architecture in several areas.

It proposes a new combined memory encryption and authentication scheme which uses very small on-chip storage area and incurs much less performance overhead compared with prior work. In addition, the thesis studies the issues of applying architectural support for data security to distributed shared memory systems. It presents a scheme which is scalable with large-scale systems and only introduces negligible performance overhead for confidentiality and integrity protection. Furthermore, the thesis also investigates another source of reducing performance overhead in secure systems through optimizing on-chip caching schemes and minimizing off-chip communications.

CHAPTER I

INTRODUCTION

Computer manufacturers are becoming increasingly concerned for the security of the computer systems they make in recent years. Meanwhile, computer users are more and more aware of the security features available on their computers as well. As computing matures further, it is expected that security will join performance and power as a key distinguishing factor in computer systems.

There are many good reasons why data security concerns are becoming very important. First, businesses need strong protection for data privacy and integrity. As more and more businesses now adopt computers as the main venue for processing their business transactions and storing critical business information, attacks in the cyber space are becoming more common at the same time due to the fruitful rewards if the attack succeeds. Many businesses hold trade secret which worth millions of dollars or even more. Their businesses are running with the premise that the confidentiality of their trade secret is secured. For example, source code at software companies, drug ingredients at pharmaceutical companies and financial data at most companies are all valuable trade secrets of their businesses. Other than the data confidentiality, the data integrity also receives intensive attention. A spoofed transaction can cause a significant loss to a financial organization or import/export company. In recent years, as computer usage is becoming more prevalent and versatile in businesses, it also provide attackers more opportunities for information theft and spoofing.

Second, the ongoing effort of combating piracy needs the support from secure systems. Software piracy causes a loss in revenue for more than 30 billion dollars annually worldwide. In this scenario, the owners of the computer systems are often the beneficiaries of the piracy, so they are not motivated to enforce the integrity of that software. As a result, achieving software integrity protection on consumer platforms is very challenging. For examples, most game consoles nowadays are built with a verification system to verify whether the software

running on the systems is authentic. However, owners of the game consoles can easily get electronic devices such as Mod-Chips to bypass the authentic software verification at a fairly low price. Obviously, the security support on these existing game consoles is not sufficient. If these systems are equipped with a better security protection scheme, it will help increase the cost of running pirated software on the game consoles and restrain the usage of pirated software. As a result, software vendors will receive a more reasonable share of the profit.

Third, utility computing requires high assurance of data security. A growing use of large-scale systems is in the context of *utility* or *on-demand computing* where a company owning large systems will lease computational and storage resources of the system to businesses which outsource their IT operations. There are many benefits for utility computing including no initial cost for hardware, higher return on capital and better concentration on core businesses. However, concerns about the privacy and integrity of data hinder the growth of utility computing. IndustryWeek pointed out that data privacy is one of the major concerns that prevent fast adoption of the on-demand computing model. With better security support, it will help relieve the security concerns for the customers of utility-computing and help the customers to realize their full potentials.

Fourth, online gaming requires secure systems to ensure fairness among the players. Online gaming is becoming a bigger and bigger business every day. A major challenge for online game operating companies is to guarantee the fairness in the game to all players. It will be of the players interest to run unauthorized software which helps them gain competitive advantage against other game players. This scenario is very similar to the previous scenario for combating software piracy, where the device owners are motivated to launch hardware based attacks to bypass the data integrity verification system in the devices.

Lastly, computer users need better security support for protecting personal privacy. Computers are carrying more and more duties in everyday life. More computer users now pay their bills, manage their financial information and do their tax return on computer, which at the same time make the computer a more popular target for information theft. Moreover, with the trend of mobile computing and pervasive computing, computer users carry their computer systems to more places such as libraries and cafes. Mobile computing

devices bring great convenience to users by making information available at fingertips. However, it also makes personal information stored on computers more vulnerable to attacks on the other hand due to its usage in the public places.

1.1 Threat Models

Better security support is of interest for almost everyone including hardware manufacturers, software companies, media providers, business owners and personal users. While many security software packages are available, they are *ineffective* in protecting against an emerging class of security attacks that involve directly tampering with the physical operation of the system. These hardware attacks may involve inserting a device on the communication path between the microprocessor and other chips to *passively* snoop their communication [12, 13], or to *actively* modify data communication between the processor and other chips [20].

All scenarios we described previously are subject to hardware attacks. In the case of protecting data confidentiality and integrity in a business, malicious hardware devices can be plugged in by company visitors or third-party vendors within just a few minutes. Clearly, software protection schemes cannot detect such devices or protect against hardware attacks. To make situation even worse, program variables of the protection software itself are normally stored in main memory and subjected to hardware attacks as well.

In the case of combating software piracy and ensuring online game fairness, the attacks are usually the device owners themselves. The attackers are motivated to use modchips to bypass the authenticity verification system on the game consoles. As a result, the owners are now able to run pirated software on their devices. For online gaming, the attackers would like to spoof the CPU cycles or modify the memory data to gain competitive advantage over other players in the game. Such hardware devices are available at a fairly low price and can be installed easily by computer owners.

In general, we can classify the hardware attacks into two categories: passive attacks and active attacks. We now describe these two categories in detail.

1.1.1 Passive Hardware Attack

Passive hardware attacks are usually performed with the help of devices such as bus snoopers. Attackers use these devices to eavesdrop the information which is transmitted on the communication path without modifying the information. One of the most important communication paths within the computer system is between the processor and the main memory. Due to the limited capacity of on-chip cache, processor often needs to store data in the main memory and read from it at a later time. If a bus snooper is sitting on the communication path between the processor and the main memory, it can easily pick up the critical information. Other than buses, processor inter-connects and external pins of chips are also potential targets for eavesdrop attack.

It will be very difficult for software protection schemes to detect passive hardware attack because the attack directly occurs on the physical bus. Moreover, passive hardware attack does not modify the information and software protection scheme will receive the same information stream as expected under the normal case. As a result, all information transmitted on unsecure communication paths are subject to passive hardware attacks. This type of attacks can also assist adversaries to launch more sophisticated attacks once critical data such as the security keys used in software-based security scheme are obtained through eavesdrop.

1.1.2 Active Hardware Attack

Active hardware attacks will directly modify information transmitted on unsecure communication paths through physical interference on buses, processor inter-connects and external pins of chips. Adversaries can leverage active hardware attacks to hijack or modify signals transmitted on the communication path. The mod-chips used to bypass the authenticity verification in game consoles belong to this category.

Since active hardware attacks directly modify signals and therefore may change the information stream transmitted on the communication path, software-based security scheme may be able to detect this type of attacks. However, active hardware attacks can also be executed in a stealthy way. For example, adversaries can launch a replay attack after eavesdropping a previous legitimate sequence of signals. If the data stream and the MAC

which is used to authenticate the data stream are replayed all together, the software-based security scheme may not be able to detect the replay attack in the case.

In addition to modifying data signals, active attacks can also modify or replay control signals. Such attacks are more difficult to detect by software-based security schemes and can trick the system into undesired states which make it vulnerable to subsequent attacks.

1.2 Background

Several major industrial efforts aim to provide *trusted* computer platforms which would prevent unauthorized access to sensitive or copyrighted information stored in the system and prevent unauthorized modification of such data. Unfortunately, such initiatives only provide a level of security against software-based attacks and leave the system wide open to hardware attacks [12, 13], which rely on *physically* observing or interfering with the operation of the system, for example by inserting a device on the communication path between the microprocessor and the main memory. Some of this communication path (e.g. the memory bus) is exposed outside the processor or main memory chips and can be tampered with easily [12, 13]. Such *hardware attacks* have been demonstrated to be feasible and relatively easy to perform in breaking the Digital Rights Management features in game consoles [12, 13], and in breaking the security features of a version of a secure processor DS5002FP [20].

Clearly, software protection cannot adequately protect against hardware attacks, because program variables of the protection software itself can be stored in main memory and be subjected to hardware attacks. Instead, hardware *memory encryption and authentication* has been proposed [6, 9, 10, 26, 27, 28, 43, 44, 45, 50, 55, 56]. The microprocessor industry has also offered secure processors for commercial use, including the IBM SecureBlue [14], and Dallas Semiconductor DS5002FP [30]. A secure processor assumes that data on a processor chip is secure, while off-chip data can be observed or modified by an attacker. Since snooping or manipulating on-chip transistors/wires is much more difficult than snooping or manipulating off-chip components/interconnections, combined with the fact that the chip

can be protected through the use of various manufacturing techniques such as special coating [30], the processor chip provides a reasonable security boundary. With this security boundary, in order to provide secrecy and integrity of data that is stored off-chip, data must be encrypted before it is moved off-chip, then decrypted and authenticated when it is brought back on-chip. Memory encryption seeks to protect against passive attacks on the secrecy/privacy of data and/or code. Memory authentication seeks to protect against active attacks on data integrity, such as modifications that cause programs to produce wrong results or behavior. Memory authentication is also needed to keep data and/or code secret because active attacks can tamper with memory contents to produce program behavior that discloses secret data or code [45].

Although memory encryption and authentication largely improve the accountability of the computer systems against hardware attacks, it increases the performance overhead of the system. Both business users and personal users need support for data security. However, if the support for data security is only available at a fairly expensive price, computer users would rather opt for the computer systems with better performance but less secure. So the first challenge of providing architectural support for memory encryption and authentication is to limit the performance overhead within a reasonable range. Most of the existing work proposed hardware support for memory encryption and authentication with noticeable space overhead and/or time overhead. In this thesis, we will propose our scheme which targets to reduce both the space overhead and the time overhead.

Moreover, memory encryption and authentication does not naturally guarantee the confidentiality and integrity of the data. The adversaries can still launch their attacks if the hardware support for memory encryption and authentication is not deployed correctly. In this thesis, we will discuss limitations of prior work and propose new schemes that overcome these limitations.

Finally, hardware support for memory encryption and authentication for uni-processor environment cannot be directly extended to multi-processor environment. Uni-processor schemes provide data encryption and authentication only for communication between the

processor and the main memory. However, communication among processors in the multi-processor environment is subject to hardware attacks as well and therefore needs to be protected. In this thesis, we will address the challenges for protecting processor-to-processor communication against hardware attacks. In particular, we will address the problem of how to protect point-to-point communication within distribute shared memory systems where broadcast mechanism is not available. Most existing work for architectural support for memory encryption and authentication in multi-processor environment rely on the assumption that each processor can observe every coherence transaction in the system, such as through the monitoring of the single shared bus in the system. In distribute shared memory systems, the interconnects among the processors could be in the layout of point-to-point. Therefore, the assumption in existing work may no longer hold and new techniques are needed.

1.3 Overview of this dissertation

This dissertation consists of six chapters. This chapter introduces the problem and challenges for solving the problem. Chapter 2 will present some of the concepts discussed in this dissertation. It will talk about the state of the art of architectural support for data security, as well as the limitations with the existing schemes.

We then present the architectural support we need to provide memory encryption and authentication for the single processor systems in chapter 3. It will address three problems in secure schemes for uni-processor systems. First, we discuss the hardware mechanism that we use to reduce the additional on-chip storage for security support. Second, we illustrate an efficient and secure scheme for memory authentication. Last, we propose a fix to a pitfall in existing secure architecture.

We proceed to extend the architectural support for memory encryption and authentication to distributed share-memory systems in chapter 4. Extending architectural support for data security to multi-processor systems face additional challenges which are not present in uni-processor systems. In particular, we focus on the distributed share-memory systems where no broadcast medium is available to all processors. We introduce a single-level data encryption and authentication scheme to protect the confidentiality and integrity of the

data in DSM environment.

While providing the new architectural support for data security to computer systems, we also notice that extra performance overhead in secure architecture can be reduced through on-chip cache optimization. In chapter 5, we discuss a novel selection cache allocation scheme which reduces cache miss rate for both data blocks and security related blocks such as counter values and MAC data. The new scheme helps realize less frequent memory accesses and improve system performance for both secure systems and baseline systems.

Finally, our conclusions are presented in Chapter 6.

CHAPTER II

RELATED WORK

In the previous chapter, we presented the motivation and challenges in protecting against hardware attacks. Architectural support for memory encryption and authentication are required to ensure the confidentiality and integrity of the data in the presence of such attacks. In this chapter, we will answer the following questions: What are the limitations of existing architectural support for memory encryption and authentication? How can we reduce the performance overhead due to the architectural support for memory encryption and authentication? Why is it difficult to extend the architectural support for memory encryption and authentication on single processor environment to distributed shared memory systems?

2.1 Basic Encryption and Authentication

There are many systems that are proposed to provide architectural support for tamper-resistance and copy-resistance environment. Execute Only Memory (XOM) [10, 27, 28] was one of the earliest such proposals. The XOM architecture provides protection for both data confidentiality and integrity. To protect the confidentiality of the data, XOM directly uses a block cipher to encrypt and decrypt data which are stored in the off-chip memory. The software that runs on the XOM architecture is encrypted by the software vendor. Only the processor which possesses the key can decrypt the software and run the software successfully. As a result, the XOM architecture guarantees that the software can only run on the target processor and it protect the confidentiality of the software as well. A combination of symmetric key cryptography and asymmetric key cryptography is used to ensure the security of the XOM architecture. Asymmetric key cryptography is very secure to transmit data through unsecure communication paths when it is difficult to establish prior key agreement. However, asymmetric key cryptography is usually 1000 times slower than the symmetric key cryptography. Therefore, encrypting the whole program data with asymmetric key cryptography is unrealistic. The software vendor first encrypts the software

using symmetric key cryptography with key K_s and later encrypts K_s with asymmetric key cryptography. Each XOM processor is equipped with a pair of public key K_p and private key K_{xom}. The software vendor sends the key K_s to the processor through asymmetric cryptography. Once the process decrypts the key K_s with its private key, it can use it to decrypt the program data through symmetric key cryptography.

To protect the integrity of the data, the XOM architecture appends the data blocks stored in off-chip memory with a MAC of the address and the data value pair. Including the blocks address in the MAC helps prevent adversaries from copying blocks from one memory location to another. However, the XOM architecture is insufficient to protect against replay attacks. The verification of the MAC guarantees that the data block and the MAC were generated by the processor, though it does not guarantee that it is the most recent copy of the data. The adversaries can replace the data and MAC in the off-chip memory with the ones which were stored at the same location before. When the processor retrieves the data and MAC back on-chip, it will validate the MAC and consume the data without noticing that the data and MAC are actually stale.

To address the limitation of the XOM architecture for the replay attacks, Suh et al. [50, 9] proposed Cached Hash Tree (CHTree) which uses a hash tree rooted in trusted memory to check the integrity of the data. Hash tree (or Merkle tress) [32] are often used to protect data integrity in unsecure storage. The leaf nodes in the hash tree are the original data and each inner level in the tree contains nodes which are the MACs of their child nodes. The cached hash tree scheme is also based on the hash tree while the root node of the tree is always kept on-chip. The internal hash nodes in the hash tree can either be kept in the cache or sent off-chip. When the data is brought back on-chip, the MAC will be verified until the level where the hash node is on-chip. Since the root node is always kept on-chip, the integrity of the data will always be verified and ensured. However, the whole verification process can take hundreds of cycles since the MAC verification may sometimes go up the hash tree for multiple levels.

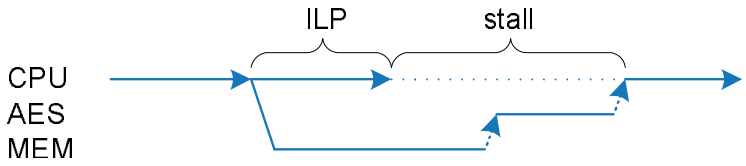
CHTree could cost significant performance overhead since it check the integrity of the memory after every memory access. To mitigate the performance overhead, several scheme

were proposed. The log hash scheme in [50] employs lazy authentication in which a program is authenticated only a few times during its execution. The memory authentication scheme used by Gassend et al. employs authentication in which instructions are allowed to commit even before their data is completely authenticated [9]. Both schemes do not verify the data integrity in a timely manner and help reduce the performance overhead since the verification process is infrequent or delayed after the actual use of the data. These schemes are referred as *lazy* authentication schemes. The Authenticated Speculative Execution proposed by Shi et al. [43] employs timely (i.e. non-lazy) authentication, but requires extensive modifications to the memory controller and on-chip caches. As pointed out by Shi et al. [45], *lazy* memory authentication sacrifices security because attacks can be carried out successfully before they are detected. Unfortunately, *timely* authentication delays some memory operations until authentication is complete. Furthermore, prior memory authentication schemes rely on MD-5 or SHA-1 to generate authentication codes but under-estimate their latency. Recent hardware implementations of MD-5 and SHA-1 show the latencies of more than 300ns [19], which is prohibitively expensive to use in timely authentication.

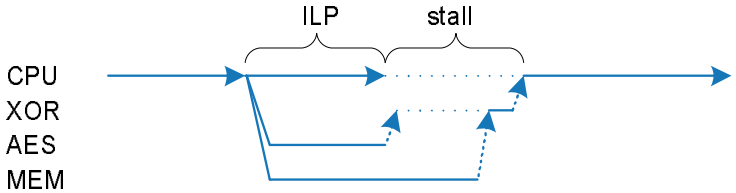
2.2 Counter Mode

In basic encryption and authentication scheme which we introduced in the previous section, an encryption algorithm such as triple DES or AES [8] is used to encrypt each cache block as it is written back to memory and decrypt the block when it enters the processor chip again [10, 27, 28]. Although this scheme is reasonably secure, direct encryption reduces system performance significantly. As shown in Figure 1(a), decryption latency of a cryptographic algorithm such as AES is added to the already problematic L2 cache miss latency. In order to reduce the performance overhead caused by basic encryption and authentication, new technique was proposed to be used in memory encryption and authentication.

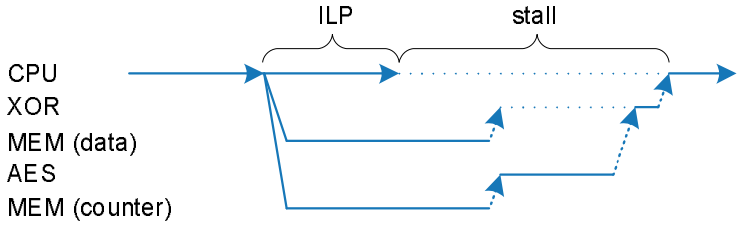
XOR-based one-time pad (OTP) cipher is a very simple yet unbreakable symmetric cipher. Unlike the encryption algorithms such as triple DES or AES which take noticeable CPU time to compute, the one-time pad cipher only takes one CPU cycle for the XOR operation. In one-time pad cipher, the sender will XOR the plaintext with a *random* pad



(a) Direct encryption.



(b) Counter cache hit in counter mode.



(c) Counter cache miss in counter mode.

Figure 1: Timeline of a L2 cache miss.

which is of the same size as the plaintext. The cipher text is generated and sent through the unsecure communication path to the receiver.

$$Plaintext \bigoplus Pad = Ciphertext$$

At the receiver's side, the ciphertext is XORed with the same pad which was used by the sender for encryption. The plaintext will be consequently generated and ready to use within a single CPU cycle of XOR operation.

$$Ciphertext \bigoplus Pad = Plaintext$$

Two requirements need to be satisfied for the one-time pad cipher to be perfectly secure. The pad used in the encryption and decryption needs to be truly random and unique for every encryption operation. If the pad is truly random, an attacker cannot compute the plaintext from the ciphertext without knowledge of the pad, even via a brute force search of the space of all possible pads! Trying all possible pads does not help crack down the one-time pad cipher at all, because all possible plaintexts are equally likely decryptions of

the ciphertext. Meanwhile, it is also essential to guarantee the uniqueness of the pads used in every encryption operation. Due to the nature of the XOR operation, the pad used in the encryption operation can be easily calculated if both the plaintext and the ciphertext are known to the attackers. Therefore, if the same pad is used in multiple encryption operations, knowledge of one plaintext/ciphertext pair will reveal the other plaintext information as well.

$$Plaintext \bigoplus Ciphertext = Pad$$

In the scenario of protecting data confidentiality and integrity for the communication path between the processor and the main memory, both the sender and the receiver are the processor. The main memory is just an unsecure temporary storage which is subject to hardware attacks. Therefore, ideally the processor should generate a random pad when it needs to send data off-chip to the main memory. The plaintext is XORed with the pad and generates ciphertext which will be stored in the main memory. The pad generated is stored on-chip until the ciphertext will be brought back on-chip and be decrypted. However, the pad is of the same size as the plaintext itself. Usually, the purpose of processor sending data off-chip is due to the storage limitation of the on-chip space. Keeping the pad on-chip will be unrealistic since it will not help at all alleviate the on-chip storage shortage. As a result, a more practicable approach, the counter mode cipher was proposed to be used in architectural support for memory encryption and authentication.

Counter mode cipher [7, 29] can be used to hide this additional AES latency [43, 44, 45, 50, 55, 56]. Instead of applying AES directly to data, counter mode encryption applies AES to a *seed* to generate a *pad*. Data is then encrypted and decrypted via a simple bitwise XOR with the pad. Although work on message encryption proves that the seed need not be secret to maintain secrecy of data encrypted in counter mode [4], it relies on a fundamental assumption that the same seed will never be used more than once with a given AES key [4, 29]. This is because the same seed and AES key would produce the same pad, in which case the attacker can easily recover the plaintext of a memory block if the plaintext of another block (or a previous value of the same block) is known or can be guessed.

The seed for memory encryption can be obtained by concatenating the data block address with a per-block counter. The address part ensures that different locations are not encrypted with the same seed. The counter for memory block is incremented on each write-back to that block to ensure that the seed is unique for each write-back. The counter value used to encrypt a block is also needed to decrypt it, so a counter value must be kept for each memory block. However, write-backs of a block could be frequent and counters can quickly become large. Thus, per-block counter storage must either be relatively large or a mechanism must be provided to handle counter overflow. In prior work [9, 43, 50, 55, 56], when a counter wraps around, the AES encryption key is changed to prevent re-use of the same pad. Unfortunately, because the same encryption key is used to encrypt the entire memory, the entire memory must be re-encrypted with the new key. With a large memory, such re-encryption consumes considerable time and "freezes" the system for a noticeable time. For example, re-encryption of 4 GB of memory at a rate of 6.4 GB/second freezes the system for nearly one second. This is inconvenient for interactive applications and may be catastrophic in real-time systems. If small counters are used, such re-encryptions will occur frequently and become a significant performance overhead.

Using a larger sequence number can reduce the frequency of these "freezes", but they degrade counter mode memory encryption performance. Counters are too numerous to keep them all on-chip, so they are kept in memory and cached on-chip in the *counter cache*, also called *sequence number cache* (SNC) in other studies. Counter mode hides the latency of pad generation by finding the block's counter in the counter cache and beginning pad generation while the block is fetched from memory, as in Figure 1(b). However, a counter cache miss results in another memory request to fetch the counter and delays pad generation as in Figure 1(c). A counter cache of a given size can keep more counters if each counter is small. As a result, a compromise solution is typically used where the counters are small enough to allow reasonably good counter cache hit rates, yet large enough to avoid very frequent re-encryptions. However, this compromise still results in using larger counter caches and still suffers from occasional re-encryption freezes.

Unlike counter secrecy, which is unnecessary [4], undetected malicious modifications of

counters in memory are a critical concern in counter mode memory encryption because they can be used to induce pad reuse and break the security of memory encryption. Although prior work addresses this concern to some extent through memory authentication, we find a flaw that has previously been ignored or unnoticed. For example, the adversaries can rollback the counter values stored in the main memory. Later, the stale counter values will be fetched back on-chip by the processor without noticing that the counter values have been modified if no verification of the counter value is presented. The processor will be tricked into using the stale counter values and applying the same pads as the ones used in previous encryption operations. This behavior breaks the non-repetition premise of the counter mode cryptography. We also find that this flaw can be avoided by authenticating the counters involved in data encryption, without a significant impact on performance.

2.3 Multi-processor System

Other than schemes of architectural support for data security in uni-processor environment that we discussed in previous sections, researchers have also proposed secure multiprocessors. Constructing a secure multiprocessor system by piecing together secure processors alone does not give sufficient protection because communication between processors is not automatically protected. In the uni-processor environment, processor-to-memory communication is protected, while in multi-processor environment, the processor-to-processor communication needs to be protect as well since the processor inter-connects are subject to hardware attacks. Therefore, the data transmitted among the processors need the same type of protection as processor-to-memory communication and this issue is not addressed in schemes for uni-processor environment. The main challenge in protecting processor-to-processor communication is that communicating processors must share necessary encryption information, so that a data block encrypted by one processor can be decrypted by another processor.

Shi et al. proposed a security model for symmetric multiprocessor (SMP) systems [43]. In a bus-based Symmetric Multi-Processor (SMP) system, the shared bus provides an ideal medium for sharing encryption information because all processors can observe the same

bus. Therefore, a global bus counter can be used to encrypt data transfers between processors [43], or data transmitted on the bus itself can be used to encrypt new data blocks through Cipher Block Chaining [56]. Additionally, [6] proposes a technique to authenticate a shared bus.

There is a recording stage where each processor updates a running hash based on messages seen on the bus, and a MAC-based stage where the running hash of each processor is used to ensure that each processor has seen a correct sequence of bus transactions. With each of these approaches, all processors keep track of the same encryption or authentication information, which is possible due to the shared bus, and use this information to protect the privacy and integrity of messages on the bus.

However, not all multi-processor systems enjoy the convenience brought by the shared bus. Distributed Shared Memory (DSM) multiprocessors use a point-to-point interconnects, so there is no shared communication medium that all processors can monitor. Hence, neither uniprocessor nor SMP protection schemes can be extended directly to protect DSM systems. Extending direct encryption and authentication for processor-to-processor communication would incur a very high performance overhead due to the added latencies at the sender side for encrypting data and generating MACs, and at the receiver side for decrypting data and verifying the MACs. With a recent hardware implementation showing an AES latency of 37ns and MD5 or SHA-1 over 300ns [19], this approach is either too costly or not feasible.

Alternatively, one may imagine an approach in which uni-processor counter-mode encryption is directly extended to protect processor-to-processor communication by treating processor-to-processor data transfer similarly to a processor-to-memory communication. However, this approach is problematic to support due to the need to keep the counters in both the sending and receiving processor coherent. For example, in response to an intervention to a dirty line, a processor flushes the line to the requester, and the flushed line would be encrypted by XORing it with a pad obtained by incrementing the current counter for the block. This increment would trigger invalidation of other cached copies of the same counter. In order for the receiving processor to decrypt the flushed line, it needs to obtain the new counter value for the block. It would do so by sending a read request for the

cache block that contains the counter, which eventually appears as an intervention to the sender processor. Hence, the latency for processor-to-processor communication is effectively doubled (obtain data, then its counter). In addition, the counter communication needs to be protected against tampering as well, so it requires high-latency authentication. Similar difficulties exist with maintaining coherency among nodes in the Merkle tree.

It is also clear that SMP protection cannot be extended easily for protecting DSM systems. The requirement that each processor observes all coherence transactions would be costly to support in terms of ensuring a global ordering of all transactions as well as the large bandwidth requirement needed for broadcasting each transaction to all processors. The number of processors in DSM systems is usually several times more than the number of processors in SMP systems. Enforcing a broadcast of each transaction to all processor will defeat the purpose of the point-to-point interconnect used in DSM systems.

To the best of our knowledge, only two secure architectures have been proposed which can be used for non-bus based interconnects [25, 41]. Most closely related to our work is the design in [41] which is also a protection scheme for DSM systems. In [41], the problem of encrypting the memory is divided into two subproblems. Previously proposed techniques for uniprocessors are used to encrypt data communicated between a processor and its local memory. To encrypt data sent during processor-to-processor communication, per-processor-pair counters are kept to encrypt and decrypt data sent from one processor to another. When one processor needs to send data to another, the pad shared between the two is used to encrypt and decrypt the data. After the transaction, each processor increments the shared counter and precomputes the next pad. Because inter-node communication involves two separate security mechanisms, we refer to this approach as a *two-level* approach.

The two-level approach involves a high number of cryptographic operations for remote requests, and this approach results in several inefficiencies and high performance overheads. For example, data in response to a read request that must be satisfied by a remote node's memory must be decrypted and authenticated, encrypted and signed, and then decrypted and authenticated again. In this thesis, we will propose a single-level approach which relies on a unified memory encryption and authentication model where data is only encrypted

and hashed when it is moved off of a processor's chip (either on a writeback or intervention request) and only decrypted and authenticated when it is brought on chip. Second, the approach in [41] can have difficulty scaling to large numbers of processors since communication counters and pads are maintained for each pair of processors. Each processor needs to maintain individual sets of counters and pads for all other processors. The number of all sets of counters maintained in the system is in the order $O(n^2)$ of the number of the processors in the system. Third, [41] requires an on-chip memory/directory controller, which limits its applicability. The scheme in this thesis utilizes a novel *single-level* approach that achieves lower execution time overheads and avoids the complexity of supporting two security protection schemes compared to a two-level approach.

In [25], an approach is proposed to protect communication in multiprocessors across an arbitrary interconnection network and coherence protocol. Instead of maintaining the per-processor-pair counters for communication between processors as in [41], this approach uses a global counter controller to distribute different counter ranges to processors in the system. To minimize the cryptographic delays caused by communication between processors, a keystream cache is used at the sender to move the AES latency off the critical path. The pads which are pre-computed and stored in the keystream cache can be used to encrypt blocks sent to other processors immediately. At the receivers side, a keystream pool is maintained to store the pads for all potential senders. When the encrypted data block arrives at the receiver, the data can be used shortly afterwards if the pads used for encryption can be found in the keystream pool. The main drawbacks of this approach are that very significant on-chip storage overheads are required for the various structures (e.g. a 512KB keystream pool). Also, data communications are vulnerable to replay attacks, especially if attackers can drop messages in the system.

2.4 Cache optimization

Caches are the processor's first line of defense against long latencies needed to access the main system memory. A cache is a small (and therefore fast) memory located close to the processor, and its goal is to quickly satisfy most of the processor's memory read and

write requests. Since the capacity of the cache is limited, a key cache design decision is the mechanism that determines which data blocks should be kept in the cache at any given time. Most data exhibit *locality*, where recently accessed data tend to be re-accessed in the near future. Caches exploit this property by keeping the most recently accessed data blocks. As a result, whenever requested data is not found in the cache (a cache miss), the data is fetched from main memory and a cache line is selected to keep that block. Due to limited cache capacity, this selection involves a replacement decision - which of the in-cache blocks should be removed from the cache to make room for the incoming line. All caches need an effective *replacement policy* to guide replacement decisions. However, there is also a need for an effective *allocation policy* to guide the decision of whether to insert a given incoming block into the cache or not. Unfortunately, existing caches nearly ubiquitously use a trivial allocation policy that always inserts an incoming block into the cache.

In memory encryption and authentication, the caching of the counters is the key to avoid the significant performance degradation. As we studied the caching schemes for the counters, we discovered that some of the techniques can be applied to data caching as well. We notice that caching techniques providing better performance can benefit both counter caching and as well as the normal data caching. In this thesis, we study a new cache allocation scheme. We use a predictor to predict the likelihood of the reuse for a given data block. We take a probabilistic approach to selective skip cache allocation if the data block is predicted not to be reused with a high confidence. The high penalty for "no-reuse" misprediction and a low penalty for a "reuse" prediction heavily favors cache allocation over skipping allocation. As a result, our scheme carefully balances potential performance improvement against the risk of heavy performance loss due to mispredictions. To accomplish this, we propose a novel probability estimator to predict the actual probability that a block will not be reused. We then probabilistically allocate a cache line for the block, with a probability proportional to the estimated probability of reuse. In essence, our scheme allocates cache lines less often for blocks whose predicted reuse it less likely, which helps it balance the moderate potential gain of skipping allocation against the steep penalty when the non-allocated block is reused.

Another approach proposed by Shi et al. [44] to reduce the on-chip space overhead

for counters is through counter prediction and precomputation. In [44], instead of caching the counter values on-chip, the processor leverage idle decryption engine cycles to predict off-chip counter values and precompute pads. There are two advantages of counter value prediction scheme over the normal counter caching scheme. First, counter prediction uses much less on-chip area compared with the counter caching scheme. Second, the counter prediction scheme is more effective in hiding memory fetch latency than the counter caching scheme when on-chip storage for counter values is limited. However, the counter prediction and precomputation scheme faces the same problem we illustrated in section 2.2. To maximize the security of the system, the off-chip counter values need to be authenticated when they are brought on-chip. In the case of counter prediction and precomputation, counter value verification is needed for every memory access since counter values are always kept off-chip. This could introduce high performance overhead due to the verification of the counter values. Moreover, for applications which have frequent memory accesses, it is very difficult to find idle decryption engine cycles. As a result, the precomputation of the pads will not be completed in time and the decryption latency will be added directly to the memory fetch latency.

With our proposed cache allocation scheme, we avoid using the on-chip cache for storing counter values as well through storing them with the normal data blocks in L2 data cache. Our cache allocation scheme helps improve the efficiency of on-chip cache usage and only incurs a small performance overhead for storing counter values in L2 data cache and performing cryptographic operations.

CHAPTER III

PROVIDING ARCHITECTURAL SUPPORT FOR MEMORY ENCRYPTION AND AUTHENTICATION IN UNI-PROCESSOR ENVIRONMENT

3.1 Motivation

This chapter aims to address three limitations in prior work. First, we want to reduce the storage overhead and the performance overhead for memory encryption. However, for counter mode encryption, these two factors usually go against each other. If we use small counter size and counter cache space, we reduce the storage overhead but increase performance overhead due to counter cache miss or whole memory re-encryption. If we use large counter size and counter cache space, we reduce the performance overhead but with the cost of high storage overhead. We need a new scheme which uses on-chip storage more effectively for counters.

Second, memory authentication schemes in prior work were expensive and thus timely authentication was not used for every memory access. Previous schemes had to make a tradeoff between performance and security for memory authentication. However, just like memory encryption latency can be parallelized with the memory access latency through counter mode encryption, memory authentication latency can also be parallelized with the memory access latency. We propose an efficient memory authentication scheme which makes timely authentication affordable.

Lastly, a security pitfall in counter encryption was overlooked in prior work. Although counter secrecy does not affect the confidentiality of the data, counter integrity does matter. In the case that counter integrity is not protected, the adversaries can roll back the counter values and trick the system into reusing the stale counter values. In this chapter, we propose an simple and affordable solution for this issue.

3.2 Overview

We present a new low-cost, low-overhead, and secure scheme for memory encryption and authentication. We introduce *split counter mode* memory encryption. The counter in this new scheme consists of a very small per-block *minor counter* and a large *major counter* that is shared by a number of blocks which together form an *encryption page*[1]. Overflow of a minor counter causes only an increment of a major counter and re-encryption of the affected encryption page. Such re-encryptions are fast enough to not be a problem for real-time systems. They also result in much lower overall performance overheads and can be overlapped with normal processor execution using a simple additional hardware mechanism. Our major counters are sized to completely avoid overflows during the expected lifetime of the machine, but they still represent a negligible space and counter caching overhead because one such counter is used for an entire encryption page.

The second contribution of this work is a significant reduction of memory authentication overheads, due to several architectural optimizations and our use of the combined Galois Counter Mode (GCM) authentication and encryption scheme [31]. GCM offers many unique benefits. First, it has been proven to be as secure as the underlying AES encryption algorithm [31]. Second, unlike authentication mechanisms used in prior work, GCM authentication can be largely overlapped with memory latency. Third, GCM uses the same AES hardware for encryption and authentication and GCM authentication only adds a few cycles of latency on top of AES encryption. In a recent hardware implementation [19], AES latencies of 36.48ns have been reported. This is a significant advantage compared to 300ns or more needed for MD5 or SHA-1 authentication hardware used in prior work on memory authentication. This low authentication latency, most of which can be overlapped with memory access latency, allows GCM to authenticate most data soon after it is decrypted, so program performance is not severely affected by delaying instruction commit (or even data use) until authentication is complete.

[1]Our encryption page is similar in size to a typical system page (e.g. 4KB), but there is no other relationship between them. In particular, large system pages can be used to improve TLB hit rates without affecting the size of our encryption pages.

Finally, we identify and eliminate a pitfall in all private-counter mode memory encryption schemes. In these schemes, each block has a separate counter which must be incremented each time that block is written to avoid encryption pad reuse. When this counter is stored in memory, it is subject to being rolled back by attackers, so it must be authenticated before it is used to encrypt the next write-back of the block. When the counter is used to decrypt a data block, authentication of the data block implicitly also authenticates its counter – a counter modification results in the wrong decrypted plain text of the data, which is discovered when data is authenticated. However, we observe that a counter can be displaced from the counter cache while the corresponding data block remains in on-chip data caches. On-chip data is considered safe and is not re-authenticated, so the next write-back of the data block fetches and uses the unsafe counter value from memory. This vulnerability can be prevented by explicitly authenticating counters themselves, which only incurs a small performance impact.

The rest of this chapter is organized as follows: we present our split counter scheme in Section 3.3 and GCM authentication in Section 3.4, Section 3.5 provides additional implementation details, Section 3.6 presents our evaluation setup, Section 3.7 discusses our evaluation results, and Section 3.8 summarizes our findings and conclusions.

3.3 Split Counter Mode Encryption

The choice of a counter size is a major tradeoff in counter mode memory encryption. Small and medium-size counters can overflow and cause a system "freeze", which for small counters can be frequent and cause significant performance overhead. Large counters do not overflow during the expected lifetime of the machine, but more memory is needed to store them and performance suffers because fewer counters fit in the on-chip counter cache. We note that a similar tradeoff exists in computer architecture when choosing the cache size, where large caches have good hit rates but are slow and small caches are fast but have lower hit rates. The solution to the cache size tradeoff is to use multiple levels of caches to achieve both good overall hit rates and good average hit latency.

Our solution to the problem of counter size employs the same principle. We use a *split*

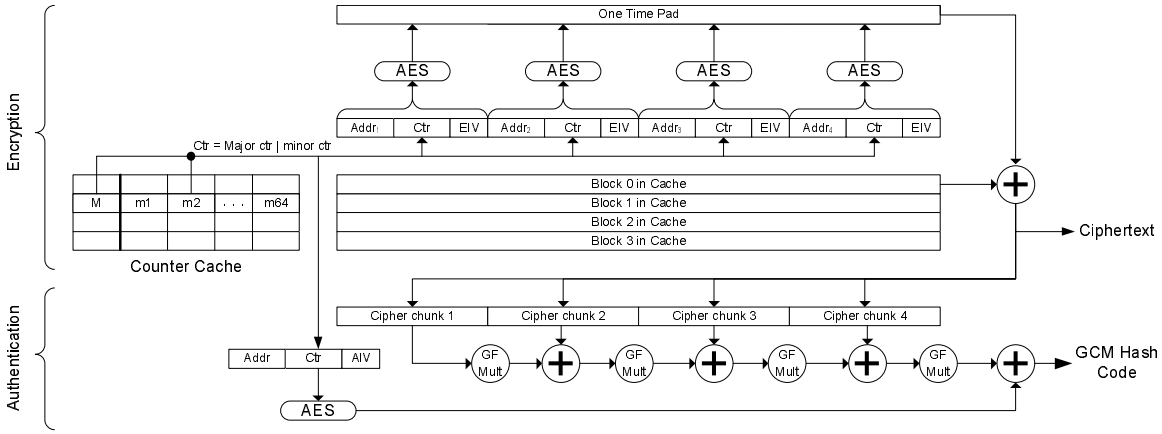


Figure 2: Split Counter Mode Memory Encryption and GCM Authentication Scheme.

counter, with a small (eight bits or less) per-block *minor counter* to reduce storage overheads and improve counter cache hit rates. We also use a large (64 bits) *major counter* that does not overflow for millennia and is shared by consecutive blocks that together form an *encryption page* which is a few kilobytes in size. Figure 2 illustrates the encryption process. When a block needs to be written back to memory, its major counter is concatenated with its minor counter to obtain its overall counter. For each encryption chunk (typically 16 byte size for AES encryption), a seed is obtained by concatenating the chunk's address, the block's counter, and a constant *encryption initialization vector* (EIV) [2]. For a 64-byte cache block size and 128-bit AES, there are four encryption chunks in a block. The encrypted chunks are then XORed with each chunk of plaintext data chunk. The figure shows that each 64-byte counter cache block stores a major counter (M) for a 4KB page and 64 7-bit minor counters ($m1, m2, \ldots, m64$) for data blocks on that page. More detail on how to choose major and minor counter sizes is provided in Section 3.5.2.

The split counter mode encryption enables several new architectural optimizations. When a minor counter overflows, we re-encrypt only its encryption page using the next major counter. Re-encryption of a relatively small encryption page is quick enough to avoid problems with real-time and interactive applications. This re-encryption is performed by fetching only those blocks which are not already on-chip. When a page is about to be

[2]The EIV can be unique per process, per group of processes that share data, unique per system, or others, depending on the needs for protection and sharing, and whether virtual or physical addresses are used.

re-encrypted, many of its blocks are already on-chip due to spatial locality and need not be fetched from memory, so this approach save bus and AES engine bandwidth. After fetching all the blocks, we change the page's major counter and zero out minor counters for all blocks on that page. The blocks are then left in the cache to be written back through normal cache replacement.

Another optimization relies on the observation that encryption pages are small, so we can track the individual state of each block in a page that is undergoing re-encryption. In particular, we track whether the block is encrypted with the old or the new major counter, which allows us to service cache accesses, misses and write-backs while re-encryption is in progress. The reduction of re-encryption work with split counters and the implementation of the mechanism for overlapping re-encryption with normal cache operation are discussed in more detail in Section 3.5.2.

3.4 Memory Authentication with GCM

Memory authentication is needed in computer systems to prevent hardware attacks that compromise data integrity, such as data modifications to crash an application or cause it to produce erroneous results. Counter-mode memory encryption also requires memory authentication to maintain data secrecy, because counters are stored in memory where an active attack can modify them (e.g. roll them back) and cause pad reuse. However, efficient, secure, and cost-effective memory authentication is difficult for several reasons. First, well-known authentication algorithms such as MD-5, SHA-1, or CBC-MAC have very long authentication latencies, and this long-latency authentication activity can begin only after data arrive on-chip. As a result, effective memory access latencies are significantly increased if data brought into the processor chip can not be used before it is authenticated, while use of data before it is authenticated presents a security risk [45]. Some use of data can safely be allowed before its authentication is complete, but only with relatively complex hardware mechanisms [43].

A second cause for the high overheads of memory authentication is the use of a Merkle tree [33]. Such a tree is needed to prevent replay attacks in which both the data and its

authentication code in memory are changed to previously observed values. Because the authentication code for the old data value matches the old value of the authentication code, the attack can remain undetected. In a Merkle tree, a leaf-level data block is authenticated by an authentication code. This code resides in a memory block which is itself authenticated by a third-level code. If K codes fit in a block, the resulting K-ary tree eventually has a root authentication code, which can be kept in a special on-chip register where it is safe from tampering. This root code, in effect, prevents undetected tampering with any part of the tree. Codes at different levels of the tree can be cached to increase authentication efficiency. If each block (of data or authentication codes) is authenticated when it is brought on-chip, its authentication must proceed up the tree only until the first tree node is found on-chip. Also, a change to a cached data or authentication code block does not need to immediately update the parent authentication node in the tree. The update can be performed when the block is written back to memory, at which time the update is propagated up the tree only until the first tree node is found on-chip. Still, a data cache miss can result in misses at all levels of the authentication tree, in which case one block from each level must be brought from memory and authenticated in order to complete authentication of the data block. The resulting bus occupancy and authentication latency can be large, while effective caching of the multiple tree levels on-chip can be difficult. Also, if data and authentication codes are cached together this can result in significantly increased cache miss rates for data accesses.

The final cause of overheads is the size of authentication codes. The probability of an undetected data modification decreases in exponential proportion to the authentication code's size, but large authentication codes reduce the arity of the Merkle tree and increase both storage and performance overheads. For example, only four 128-bit AES-based authentication codes can fit in a 64-byte block, which for a 1GB memory results in a 12-level Merkle tree that represents a 33% memory space overhead.

We address the authentication latency problem by using Galois Counter Mode (GCM) [31] for memory authentication. As illustrated in Figure 2, GCM is a counter-based encryption scheme which also provides data authentication. The encryption portion operates as a standard counter mode, by generating a sequence of one-time-pads from a seed and XORing

them with the plaintext to generate the ciphertext. In our case, the plaintext is the data block and the seed is the concatenation of the block address, the counter value, and an initialization vector. Decryption functions identically, by swapping the plaintext with the ciphertext. The authentication portion of GCM is based on the GHASH function, which computes a hash of a message ciphertext and additional authentication data based on a secret key. As shown in the lower half of figure 2, the additional authentication data input is unused in memory authentication, and the GHASH function consists of the chain of Galois Field Multiplications and XOR operations on the chunks of the ciphertext. The final GHASH output is XORed with the authentication pad, which is generated by encrypting the concatenation of the block address, the counter, and another initialization vector. The resulting hash can be clipped to any number of bits from 0 to 128, depending on the desired level of protection.

We choose to use GCM because it has been studied extensively and shown to be secure [31], and because the latency to compute hash codes can be much less than using SHA-1 or MD5. As discussed in [31], the hashed data is sufficiently obscured as to be provably secure, assuming that the underlying block cipher is secure, and that no pad is ever repeated under the same key. We meet both conditions by using the AES block cipher and by ensuring that seeds are non-repeating since they are composed of an incrementing counter and the block address. The GHASH operation in GCM can be very fast, so the latency of GCM authentication can be much less than functions such as SHA-1 or MD5 because the heavy computation (generating the authentication pad using AES) can be overlapped with loading the data from memory. Once the ciphertext has been fetched, the GHASH function can be computed quickly on the ciphertext chunks because the field multiplication and XOR operations can each be performed in one cycle [31], and the final XOR with the authentication pad can be performed immediately afterwards because this pad has already been computed. Lastly, unlike SHA-1 or MD5 which require a separate authentication engine, GCM uses the same AES engine used for encryption.

To reduce the impact of the Merkle tree on authentication latency, we compute authentication codes of all needed levels of the authentication tree in parallel. Upon a data cache

miss, we attempt to locate its authentication code on-chip. If the code is missing, we request its fetch from memory, begin generating its authentication pad, and attempt to locate the next-level code on-chip. This is repeated until an on-chip code is found. When the requested codes begin arriving from memory, they can be quickly authenticated as they arrive. Once the authentication chain from the data block to the original on-chip authentication code is completed, the block can be used safely.

Finally, we consider several options for increasing the arity of the Merkle tree and reducing the corresponding memory space, bandwidth, and on-chip storage overheads of authentication. We examine the effect of using a single authentication code for several neighboring memory blocks, as well as the effect of using smaller authentication codes. This last optimization is more effective in terms of performance, but degrades security in proportion to the reduction in authentication code size. However, we note that the need for large authentication codes was established mostly to reliably resist even a long sequence of forgery attempts. This is needed, for example, in a network environment where each forged message must be rejected, but little can be done to prevent attacks. In contrast, a few failed memory authentications tell the processor that the system is under a hardware attack. Depending on the deployment environment, corrective action can be taken to prevent the attack from eventually succeeding. In a corporate environment, a technician might be called to remove the snooper from the machine and prevent it from eventually succeeding. In a game console, the processor may produce exponentially increasing stall cycles after each authentication failure, to make extraction of copyrighted data a very lengthy process. In both cases, it is assumed that the user or the software vendor is willing to tolerate a small but non-negligible risk of a small amount of data being stolen by a lucky guess. In many environments such a risk would be tolerable in exchange for significant reduction in performance overhead and cost.

3.5 Implementation

3.5.1 Caching Split Counters

In prior work, monolithic counters are usually cached in an on-chip counter cache. In our new split counter scheme, minor counters can be kept in a similar counter cache. A seemingly obvious choice for major counters is to keep them in page tables and on-chip TLBs, but large major counters would significantly increase TLB entries and would slow down TLB lookups. We note that counters are needed only for L2 cache misses, so using a performance-critical structure like the TLB is overkill. Also, placement of major counters in the TLB ties the size of an encryption page to the size of a system page, which can result in large page re-encryption overheads when large system pages are used to reduce TLB miss rates. Another obvious choice is to keep major counters by themselves in a separate region of memory, and cache them on-chip either in a separate cache or in separate blocks of the counter cache. However, this complicates cache miss handling, as a single L2 cache miss can result in botha a major and a minor counter cache miss.

As a result of these considerations, we keep major and minor counters together in memory and in the counter cache. A single counter cache block corresponds to an encryption page and contains the major counter and all minor counters for that page. With this scheme, a single counter cache lookup finds both the major and the minor counter. If the lookup is a miss, only one block transfer from memory is needed to bring both counters on-chip. Furthermore, we find that the ratio of counter-to-data storage can be easily kept at one byte of counter to one cache block of data, which includes the storage overhead of both the minor counters and a 64-bit major counter per encryption page. An example for a 64-byte block size in the counter cache is shown in Figure 2, where a cache block stores one 64-bit major counter (M) and 64 seven-bit minor counters ($m1, m2, \ldots, m64$). If the L2 cache block size is also 64 bytes, a counter cache block corresponds to an encryption page of 4KB, because it consists of 64 blocks of 64 bytes. As another example, a 32-byte block size in both the L2 and counter caches results in a counter cache block that stores one 64-bit major counter and 32 six-bit minor counters, and an encryption page size of 1KB. Our experiments indicate little performance variation across different block sizes, because reduced per-page re-encryption

work with smaller encryption pages compensates for increased number of re-encryptions caused by smaller minor counter size.

3.5.2 Optimizing Page Re-Encryption

With monolithic counters and with our new split counters, pad reuse on counter overflow is prevented by changing another parameter used in pad generation. In regular counter mode, the only parameter that can be changed is the key, which is the same for an entire application and its change results in entire-memory re-encryption. In our split counter approach, the major counter can be changed on minor counter overflow, and this change only requires re-encryption of one encryption page.

Memory access locality of most applications is such that some blocks are written back much more often than others. As a result, some counters advance at a much higher rate than others and overflow more frequently. Consequently, some pages are re-encrypted often, some rarely, and some never (read only pages). With monolithic counters, the first counter that overflows causes re-encryption of the entire memory, so the rate of advance of the fastest-advancing counter controls the re-encryption rate for all blocks in the main memory. In contrast, with our split counters, the re-encryption rate of a block is determined by the rate of advance of the fastest-advancing counter on that page. Most pages are re-encrypted much less often than those that contain the fastest-advancing minor counter. This better-than-worst-case behavior of split counters results in significantly less re-encryption work than with monolithic counters. Our experimental results indicate that our split counter scheme with a total of eight counter bits per block (7-bit minor counters and 64-bit major counter shared by 64 blocks) on average results in only 0.3% of the re-encryption work needed when eight-bit monolithic counters are used.

In addition to performing less re-encryption work and splitting this work into smaller pieces to avoid lengthy re-encryption freezes, page re-encryption has an advantage over entire-memory re-encryption in that the re-encryption can be nearly completely eliminated from the processor's critical path. Conceptually, re-encrypting a block is a two-step process where the block is first decrypted by fetching it on-chip, and is encrypted again with a

new major counter by writing it back to memory. In our page re-encryption, the first step (fetching blocks on-chip) only needs to be performed for those blocks that are not already on-chip. Additionally, the second step (writing blocks back) does not require replacing already-cached blocks immediately. Since such blocks are likely still needed, we simply set such blocks to a dirty state, and let them be written back when they are naturally replaced from the cache. After the major counter for the page is changed and the minor counters zeroed out, write-backs of such blocks will encrypt them with the new major counter. As a result of this "lazy" approach, re-encryption of on-chip blocks requires no extra memory reads or writes. Our experimental results indicate that, on average, about half (48%) of the page's blocks are present on-chip when the page re-encryption is needed, which nearly halves the re-encryption latency and its use of memory, bus, and AES bandwidth. In contrast, in entire-memory re-encryption the blocks that are cached on-chip constitute only a small fraction of the main memory, and therefore do not noticeably reduce the re-encryption work.

Finally, our encryption pages are small enough to permit tracking of the re-encryption status of each block within a page. Such tracking allows normal cache operation to proceed during page re-encryptions and nearly completely hides re-encryption latency. To accomplish this, our processor maintains a small number (e.g. eight) of *re-encryption status registers* (RSRs). Each RSR has a *valid* bit that indicates whether it is in-use or free. An RSR is tagged with an encryption page number, and it stores the *old major counter* for the page. An RSR corresponding to a page also maintains a *done* bit for each block on that page, to indicate whether the block has already been re-encrypted. Re-encryption of a page begins by finding a free RSR (with a zero valid bit), setting its valid bit to one, tagging it with the page's number, copying the old major counter into the RSR, clearing all the done bits in the RSR, and incrementing the major counter in the counter cache. The RSR then issues requests to fetch the blocks of the page that are not already cached. As each block arrives from memory, the RSRs are checked. If the block's page matches an RSR and the block's *done* bit is not set, the block is decrypted using the old major counter from the RSR. Then the block's minor counter is reset, the *done* bit in the RSR is set, and the block is supplied to the cache and its cache state is set to dirty. To avoid cache pollution

from blocks that are fetched by the RSR from memory, they are immediately written back. Any write-back, regardless of whether it is cache-initiated or RSR-initiated, is performed normally, using the block's minor counter and its page's major counter from the counter cache. This completes re-encryption of a block if its page is being re-encrypted.

Whenever a *done* bit is set in the RSR, we check if any of its done bits are still zero. If no zero *done* bit remains, all of the page's blocks have been re-encrypted and the RSR is freed by setting its valid bit to zero. To avoid re-encryption fetches of blocks that are already in-cache, the RSR looks up each block in the L2 cache before requesting the block from memory. If the block is already cached, the block's *done* bit is set immediately without creating a request to fetch it from memory.

With this support, because the cache can continue to satisfy regular cache requests even for blocks in pages that are still being re-encrypted, the processor is not stalled due to re-encryptions. A cache read or write request to a block in a page that is being re-encrypted will find the block either already re-encrypted with its *done* bit set to one, or find it being fetched by the RSR. In the latter case, the request can simply wait until the block arrives. Similarly, regular cache write-back of a block in a page that is being re-encrypted can proceed normally using the new major counter for the page.

We note that this last optimization would be difficult to achieve for entire-memory re-encryption, because it would be very costly to track the individual re-encryption status of the very large number of blocks involved in entire-memory re-encryption. In our split counter approach, however, all three optimizations can be applied relatively easily to completely avoid system freezes on re-encryptions and eliminate nearly all of re-encryptions' performance overhead.

In our scheme, cache operations can stall only when a write-back of a block causes another minor counter overflow while the block's page is still being re-encrypted, or when an RSR cannot be allocated because all RSRs are in use. The former situation is easily detected when a page's RSR allocation request finds a matching valid RSR, which can be handled by stalling the write-back until the RSR is freed. With a sufficiently large minor counters (larger than 4 bits), we find that the situation does not occur because a page

re-encryption can be completed long before a new minor counter overflow triggers another re-encryption. The latter situation is also handled by stalling the write-back until an RSR becomes available. With a sufficient number of RSRs (e.g. 8), we find that the situation does not occur because there are typically very few pages that are being re-encrypted at the same time. Consequently, RSRs only introduce very small storage overheads of less than 150 bytes. Finally, RSR lookups do not introduce much overhead because in most cases it can be performed in parallel with cache misses.

3.5.3 Protecting Data and Counter Integrity

As proven by [4], data secrecy can be maintained even if the counters in counter-mode encryption are themselves stored unencrypted. However, counter integrity must be protected because undetected counter tampering, such as rolling back the counter to its old value, may lead to pad reuse. We call such attacks *counter replay attacks*.

Protection of data integrity can help maintain counter integrity indirectly. The block's counter is used to decrypt the block which is then authenticated, and in GCM the counter is directly used in authentication of its data block. Because authentication would fail for a data block whose counter has been modified, we say that the counter is *indirectly authenticated* when the corresponding data block is authenticated.

However, we observe that a data block may reside on-chip while its counter is displaced from the counter cache to memory. When the data block is written back, the counter value from memory may have been modified and should be authenticated before it is incremented and used to encrypt the block.

To ensure secrecy and integrity of the data, we build a Merkle tree whose leaf-level contains both the data and its *direct counters*. These are the counters directly involved in encryption and authentication of data blocks. Since the split counters in our scheme are small, the overhead for incorporating direct counters into the Merkle tree is also small. With GCM, in addition to direct counters, we need *derivative counters* which are used in authentication of hash-code block in the tree. Since derivative counters are only used for authentication, data secrecy cannot be compromised by compromising the integrity of these

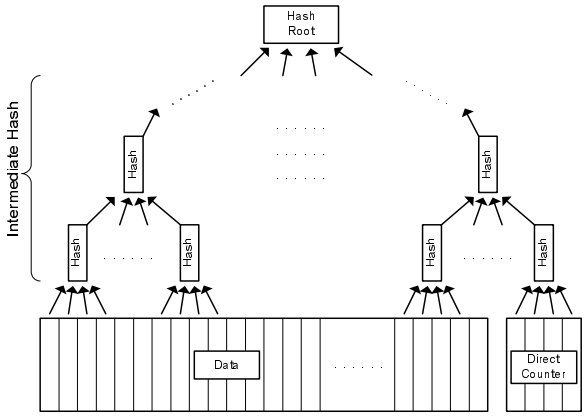


Figure 3: Merkle Tree.

counters.

Figure 3 shows the resulting Merkle tree. The on-chip hash root guarantees the integrity of the data, the direct counters, and the other hash codes in the tree.

3.5.4 Other Implementation Issues

We list the other implementation issues as follows:

Overflow of Major Counters. Our results in Section 3.7.1 indicate that 64-bit counters are enough for many millennia of overflow-free execution. A block's counter is incremented only when the block is written back, so all memory system bottlenecks (such as bus bandwidth) also limit the rate of counter increase. If we assume 64-byte cache blocks, a futuristic bus with a 16GB/s bandwidth, a bus completely saturated with write-back traffic for only one block, and that each of these write-backs is a minor counter overflow, a 64-bit counter would overflow only after more than two thousand years. With this in mind, we assume that major counter overflow will never occur within the useful life of the system.

Dealing with Shared Data. In a multi-processor environment or for memory-mapped pages shared between the processor and I/O devices, data may be shared by more than one entity. For multiprocessor environment, in addition to protecting the confidentiality and integrity of processor-memory communication, we need to protect processor-processor communication as well. We will discuss in details in next chapter about how to extend our split counters and GCM scheme to large-scale multiprocessor environments.

Virtual vs. Physical Address. The block address that is used as a component of the block's encryption seed can be a virtual or physical address. Virtual addresses are more difficult to support because they are not directly available in the lowest level on-chip cache, and different processes may map the same location at different virtual addresses. Physical addresses are easier to use but require re-encryption when memory is paged from or to the disk. Our split counters and GCM mechanisms are orthogonal to these issues, and are equally applicable when virtual or physical addresses are used.

Key Management and Security. We assume a trusted operating system and a scheme that can securely manage keys and ensure they are not compromised. Our contributions are orthogonal to the choice of a key management scheme and a trusted platform, and can be used to complement the platform's protection against software attacks with low-cost, high-performance protection against hardware attacks and combined hardware-software attacks.

3.6 Experimental Setup

Simulator. We use the SESC execution-driven cycle-accurate simulator [15] to simulate a three-issue out-of-order processor running at 5GHz. The processor is equipped with L1 instruction and data caches of 16KB each, and with a unified 1MB L2 cache. For counter-mode encryption and GCM, the processor also contains a 32KB counter cache. L1 caches are 4-way set-associative, and the L2 cache and the counter cache are 8-way set associative. The block size is 64 bytes in all caches. A block of our split counters consists of 64 7-bit minor counters and one 64-bit major counter, for a total block size of 64 bytes and an encryption page size of 4KB. The simulated processor-memory data bus is 128bits wide and runs at 600MHz, and below the bus the uncontended round-trip memory latency is 200 processor cycles. The 128-bit AES encryption engine we simulate has a 16-stage pipeline and a total latency of 80 processor cycles. This is approximately twice as fast as the AES engine reported in [19], to account for future technological improvements. The SHA-1 authentication engine is pipelined into 32 stages and has a latency of 320 processor cycles. This is more than 4 times as fast as reported in [19], to account for future

technological improvements and other developments that might give it an advantage over GCM authentication that uses the AES engine. The default authentication code size is 64 bits, and we assume a 512MB main memory when determining the number of levels in Merkle trees. In addition to authenticate the program data, we authenticate the counters as well to protect counter integrity. To handle page re-encryptions in our new split-counter mode, the processor is equipped with 8 re-encryption status registers (RSRs). Numerous other parameters (branch predictor, functional units, etc.) are set to reflect an aggressive near-future desktop machine, and all occupancies and latencies are simulated in detail.

Benchmarks. We use SPEC CPU 2000 benchmarks [47] with reference input sets. We only omit Fortran 90 benchmarks because our simulator infrastructure does not support that language at this time. Execution of each benchmark is fast-forwarded for 5 billion instructions and then simulated it for 1 billion instructions, to skip initialization and get representative results.

3.7 Evaluation

3.7.1 Split Counter Mode

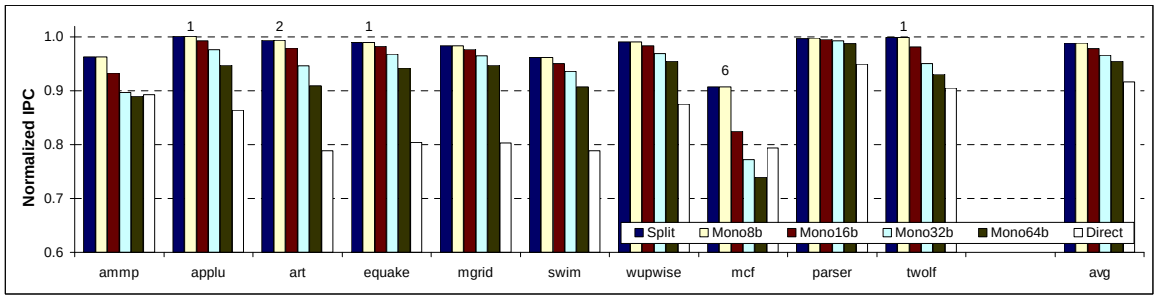


Figure 4: The IPC of different memory encryption schemes, normalized to a system with no memory encryption.

Figure 4 compares the IPC of our split counter mode memory encryption (*Split*) with direct AES encryption (*Direct*) and with the regular counter mode that uses 8-bit, 16-bit, 32-bit and 64-bit counters (*Mono8b*, *Mono16b*, *Mono32b*, and *Mono64b*, respectively). No memory authentication is used, to isolate the effects of different encryption schemes, and results are normalized to the IPC of a processor without any memory encryption support. In the figure, we only show those applications with more than a 5% performance penalty

Table 1: The counter growth rate comparison.

Apps	Counter Growth Rate (per second)				
	Mono8b	Mono16b	Mono32b	Mono64b	Global32b (million)
applu	2090	2075	2035	1961	17.2
art	2039	2010	1943	1866	17.8
equake	1323	1314	1307	1272	3.2
mcf	1211	1101	1031	987	20.3
twolf	1079	1059	1026	1005	4.5
avg	633	626	577	596	5.9

on direct AES encryption. The average is, however, calculated across all benchmarks.

In our 1-billion-instruction simulations (less than one second on the simulated machine), we observe counter overflows only in *Mono8b* configuration and overflow of minor counters in the *Split* configuration. Page re-encryptions in the *Split* configuration are fully simulated and their impact is included in the overhead shown in Figure 4. For *Mono8b*, we do not actually simulate entire-memory re-encryption, but rather assume it happens instantaneously and generates no memory traffic. However, we count how many times entire-memory re-encryption occurs and show the number above each bar. Note that our *Split* configuration with 7-bit minor counters and fully simulated page re-encryption has similar performance to the *Mono8b* configuration with zero-cost entire-memory re-encryption. From this we conclude that our hardware support for page re-encryption succeeds in removing the re-encryption latency from the processor's critical path.

To estimate the average time between entire-memory re-encryptions, we track the growth rate of the fastest-growing counter in each application as shown in table 1. The counter growth rate is measured as the number of write-backs per second for the fastest growing counter. We use these growth rates to estimate the interval between consecutive entire-memory re-encryptions with monolithic counters. The first four schemes in the tables use *locally incremented* counters, which are incremented when the corresponding data block is written back. It is also possible to use a *globally incremented* counter for encryption, where a single global counter is stored on-chip, incremented for every write-back, and used to

Table 2: The estimated time for counter overflow comparison.

Apps	Est. Time for Counter Overflow				
	Mono8b (seconds)	Mono16b (minutes)	Mono32b (days)	Mono64b (millennia)	Global32b (minutes)
applu	0.1	< 1	24	298,259	4
art	0.1	< 1	26	313,395	4
equake	0.2	< 1	38	459,914	22
mcf	0.2	1	48	592,417	4
twolf	0.2	1	48	581,975	16
avg	0.4	2	86	981,417	12

encrypt the block. We note that the counter value used to encrypt the block must still be stored separately for each block so that the block can be decrypted. However, use of a global counter would eliminate the vulnerability we discuss in Section 3.5.3 without the need for our proposed authentication of direct counters.

Table 1 shows the growth rate for the five applications with fastest-growing counters (applu, art, equake, mcf, and twolf). Averages across all benchmarks are also shown. We note that the growth rate decreases as counters become larger. This is the effect of lowered IPC with larger counters: the *number* of counter increments is nearly identical for all counter sizes, but with larger counter sizes the execution time is longer and the resulting counter increase *rate* is lower.

The global counter grows at the rate of write-backs in each application. In table 2 we compare the estimated time to counter overflow between global counter scheme and private counter scheme. With the 32-bit counter size, the global counter overflows in 12 minutes on average, much more frequently than the 32-bit private counter scheme. We also noticed that although equake and twolf are among the top 5 for locally incremented counter growth rate, their numbers of write-backs per second are below the average. This is because these two applications have relatively small sets of blocks that are frequently written back, but the overall write-back rate is not very high.

Although few entire-memory re-encryptions were observed during the simulated one billion instructions in each application, we see that the counter overflow problem is far from

negligible. Small 8-bit counters overflow up to ten times per second in some applications and every 0.4 seconds on average. Larger 16-bit counters overflow at least once per minute in some applications and every two minutes on average. Even 32-bit counters overflow more than once per month in some applications (applu and art in our experiments), which can still be a problem for real-time systems that cannot tolerate the freeze caused by an entire-memory re-encryption. We note, however, that 64-bit counters are free of entire-memory re-encryptions for many millennia.

With our split counters, we achieve the best of both worlds in terms of performance and counter overflow: small per-block minor counters allow good counter cache hit rates and good performance (Figure 4), while large per-page major counters prevent entire-memory re-encryptions for millennia (Table 2).

To help explain the performance of our split counter mode, we track the number of data cache blocks that are already resident on-chip when a page re-encryption is triggered. On average across all applications, we find that 48% of the blocks are already on-chip, which proportionally reduces the re-encryption work and overheads. The average time used for a page re-encryption is 5717 cycles. Note that normal processor execution continues during this time, even when multiple (up to three) page re-encryptions are in progress.

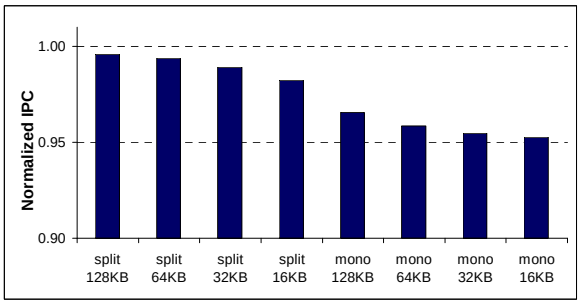
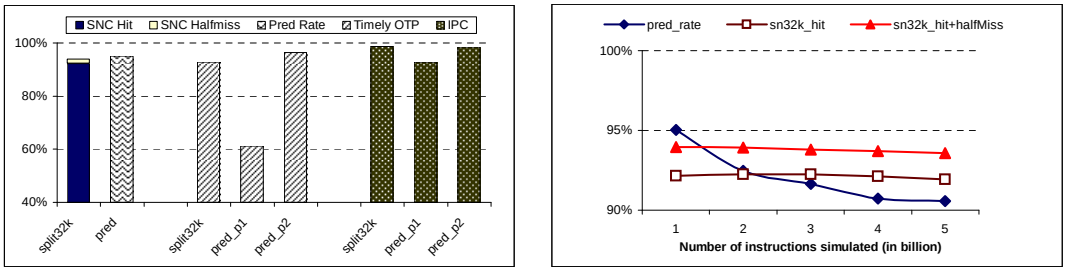


Figure 5: The IPC of different memory encryption schemes and different counter cache sizes, normalized to the system with no memory encryption.

We repeat the experiments from Figure 4 while varying the size of the counter cache from 16KB to 128KB. For the regular counter mode, we use 64-bit counters which do not cause entire-memory re-encryptions and system freezes. The averages from this experiment are shown in Figure 5. We see that, even with a 16KB counter cache, our split counter encryption (*split 16KB*) outperforms monolithic counters with 128KB counter caches (*mono*

128KB). The two schemes can keep the same number of per-block counters on-chip and have similar counter cache hit rates, but the *split 16KB* scheme consumes less bandwidth to fetch and write back its smaller counters.



(a) Performance. (b) Trend.

Figure 6: Compare split counter mode with OTP prediction and precomputation

Figure 6 compares the split counter mode with the counter prediction and one-time-pad (OTP) precomputation scheme proposed in [44]. We note that the counter prediction scheme eliminates on-chip caching of its large 64-bit per-block counters, but they are still stored in memory and represent a significant overhead (e.g. with 64 bits per 64-byte block, the overhead is 1/8 of the main memory). Moreover, this prediction involves pre-computing N pads with predicted counter values. This improves the prediction success rate, but increases AES engine utilization N-fold. We use N=5 in our experiments, as recommended in [44].

In Fig 6(a), the first group of bars shows the hit rate and half miss rate for the sequence number cache and the prediction rate for the counter prediction scheme. The counter prediction rate in the prediction scheme is better than the counter cache hit rate in our scheme, but not by a large margin.

The second group of bars in Figure 6(a) shows the percentage of timely OTP pre-computations for memory read requests. Pred_p1 represents the counter prediction scheme that uses a single AES engine. Pred_p2 is the counter prediction scheme with two AES engines, and split32k is our split-counter scheme with a 32kByte counter cache and one AES engine. Because it pre-computes five different pads for each block decryption, counter prediction requires significantly more AES bandwidth and, with only one AES engine, generates pads on time for only 61% block decryptions. With two AES engines, counter

prediction generates timely pads for 96% of block decryptions which is slightly better than the timely-pad rate of our scheme. We note that the area overhead for a deeply-pipelined AES engine could be quite significant [19].

The third group of bars in Figure 6(a) shows the average normalized IPC. The Pred_p2 scheme keeps large 64-bits counters in memory and fetches them with each data block to verify its predictions. The additional memory traffic offsets the advantage it has in terms of timely pad generation, and results in nearly the same performance as our split-counter scheme.

Figure 6(b) shows the trend of counter prediction rates in the counter prediction scheme and counter cache hit rate in our split counter scheme. As the application executes, our counter cache hit rate remains largely unchanged. In contrast, the counter prediction rate starts off with a high prediction rate, because all counters have the same initial value and are easily predicted. However, as counters change in value at different rates, their values become less predictable.

Note that our simulation results do not conflict with result reported in [44], where an extremely deeply pipelined AES engine is used to achieve very high AES bandwidth. Our additional experiments also confirm that the counter prediction scheme with two AES engines outperforms the Monolithic counter scheme with 64-bit counters. However, our split counter scheme with a 32kByte counter cache holds the same number of counters as a 256kByte cache with large monolithic counters, and needs far less bandwidth to fetch and write back its small counters.

3.7.2 GCM Authentication

Figure 7 compares the IPC of our GCM memory authentication (*GCM*) with SHA-1 memory authentication whose latency we vary from 80 to 640 cycles. No memory encryption is used, to isolate the effects of different authentication schemes, and all results are normalized to the IPC of a processor without any support for memory authentication. Note that since no counter-mode encryption is used, only GCM maintains per-block counters needed for its authentication. As before, the average is for all benchmarks, while we omitted a few

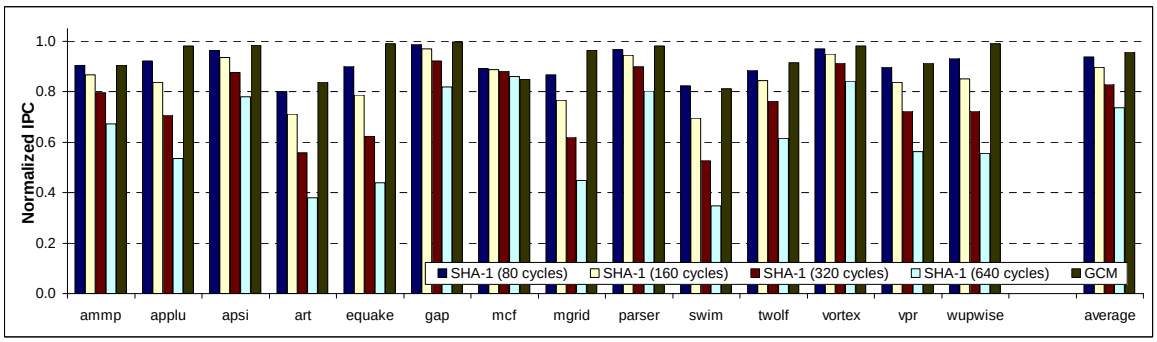


Figure 7: The IPC of different memory authentication schemes without memory encryption, normalized to a system with no memory encryption and authentication.

benchmarks with a small IPC degradation caused by authentication for clarity.

We observe that, in almost all cases, our GCM authentication scheme performs as well or slightly better than SHA-1 authentication even when assuming an unrealistically low latency of 80 cycles for the SHA-1. As the latency of SHA-1 is increased to a more realistic value, the benefit of GCM authentication becomes significant, especially in *applu*, *art*, *equake*, *mgrid*, *swim*, and *wupwise*. On average we observe that GCM authentication results in only a 4% IPC degradation, while SHA-1 with latencies of 80, 160, 320, and 640 cycles reduces IPC by 6%, 10%, 17%, and 26% respectively. The only case where GCM authentication performs relatively poorly is in *mcf*. The reason behind this is that GCM authentication uses counters, which add additional bus contention when the counter cache miss rate is significant.

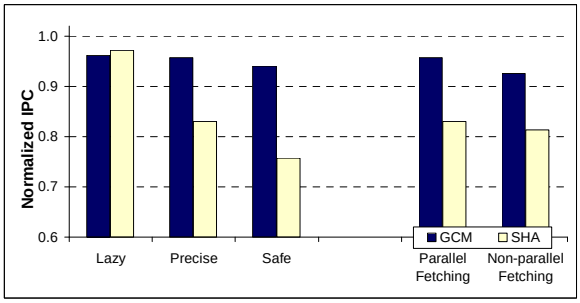


Figure 8: The IPC for GCM and SHA-1 with different authentication requirements, normalized to a system with no memory authentication.

To determine how security requirements affect performance of authentication, in Figure 8 we show the results for our GCM scheme and SHA-1 (with the default 320-cycle latency)

when we use *Lazy* authentication in which application continues without waiting for authentication to complete, *Commit* authentication in which a load instruction that misses in the data cache cannot retire until its data has been authenticated, and *Safe* authentication in which load execution is delayed on a cache miss until authentication is complete.

With *Lazy* authentication, because instruction retirement does not wait for the result of authentication, the latency of authentication is largely irrelevant. Thus, although its latency is lower, GCM has a slightly worse performance due to bus contention for counter fetches and write-backs. However, as discussed in Section 3.4, this type of authentication presents a security risk, so a more strict form of authentication is desired. With *Commit* or *Safe* authentication, the latency of authentication become important and GCM has a considerable performance advantage in most cases. Even the strictest *Safe* authentication, which with SHA-1 results in a 24% IPC reduction, results in a tolerable 6% IPC reduction with GCM.

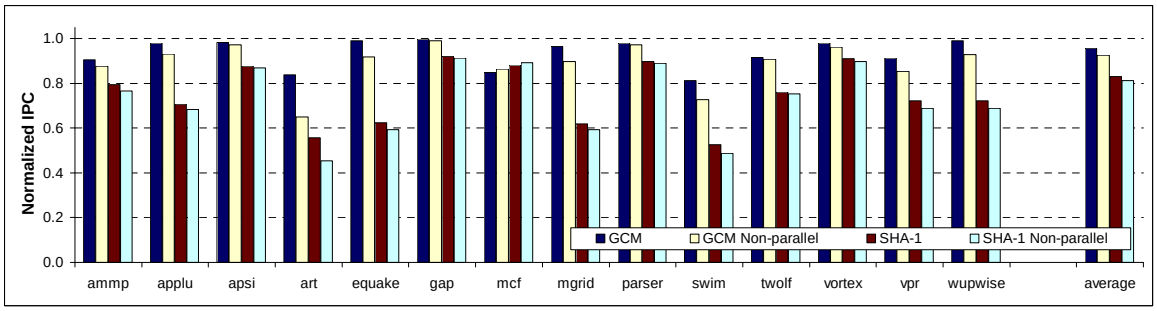


Figure 9: The IPC comparison of parallel and non-parallel GCM and SHA-1 authentication schemes, normalized to a system with no memory authentication.

Figure 9 compares parallel authentication of all off-chip Merkle tree levels on a cache miss against authentication of the next tree level only when the previous level has been authenticated. Parallel authentication provides an IPC increase for both GCM and SHA-1, and in *applu*, *art*, *equake*, *swim*, *vpr*, and *wupwise* this benefit is considerable. On average, parallel authentication of tree levels provides an average 3% IPC increase with GCM and a 2% increase with SHA-1. Although the IPC benefit seems modest, in terms of overhead reduction it is significant – in GCM, the IPC overhead of memory authentication is nearly doubled when parallel tree-level authentication is not used.

3.7.2.1 Sensitivity Analysis

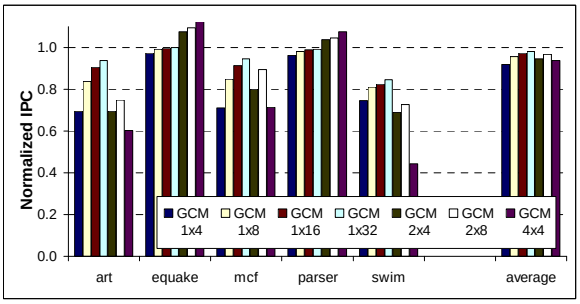


Figure 10: The IPC of GCM authentication with different parameters, normalized to a system with no memory authentication.

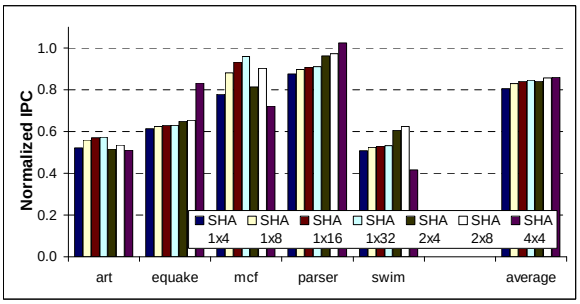


Figure 11: The IPC of SHA-1 authentication with different parameters, normalized to a system with no memory authentication.

We repeat the experiments from Figure 7 while varying the size of authentication codes and the number of data blocks that are authenticated together using the same authentication code. These variations are shown as MxN, where M is the number of data blocks that share the same authentication code, and N is the number of authentication codes that fit in a cache block. Our default scheme is 1x8 because it uses one code per data block and four 64-bit codes fit in a 64-byte line. We show these results for our GCM authentication (Figure 10) and for SHA-1 (Figure 11).

The results show us that, as we decrease the size of authentication codes from 16B (*GCM 1x4*) to 2B (*GCM 1x32*), the overhead of authentication decreases because more authentication codes fit in a cache block. This flattens the resulting Merkle Tree and reduces the number of authentication code fetches. As we increase the number of data blocks that share an authentication code, two opposite effects can be seen. In some applications such as *equake*, *parser*, and *wupwise*, this creates a prefetching effect and IPC improves because a

cache miss for one data block results in fetching its neighbors to authenticate them together. However, other applications do not benefit from the prefetching effect, but suffer from the extra contention for cache space and for the bus bandwidth.

3.7.3 GCM and Split-Counter Mode

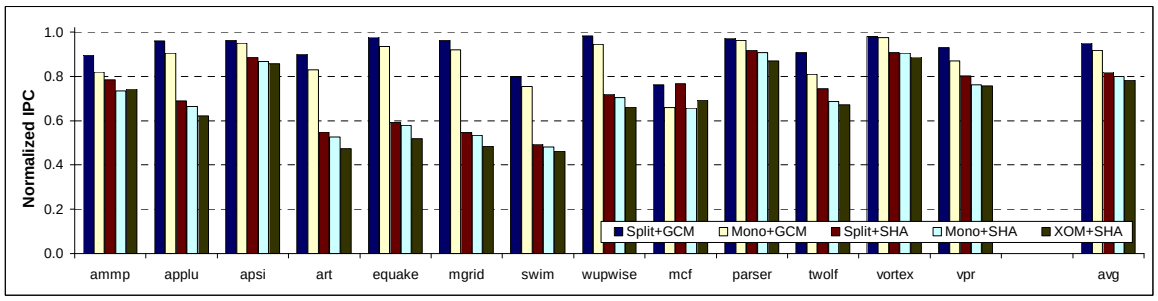


Figure 12: The IPC comparison for different memory encryption and authentication scheme combinations, normalized to a system with no memory encryption and authentication.

Figure 12 shows our results when we use both memory encryption and memory authentication. Our combined GCM encryption and authentication scheme with split counters is shown as *Split + GCM*. We compare this scheme to a scheme that uses GCM with monolithic counters (*Mono + GCM*), a scheme that uses split-counter mode encryption with SHA-1 authentication (*Split + SHA*), a scheme that uses monolithic counters and SHA-1 authentication (*Mono+SHA*), and a scheme that uses direct AES encryption and SHA-1 authentication (*XOM+SHA*). As before, all IPCs are normalized to a system without any memory encryption or authentication. Only the benchmarks with significant differences among the schemes are shown, but the average, as before, is for all the benchmarks. Our combined GCM mechanism with split counters results in an average IPC overhead of only 5%, compared to the 20% overhead with existing monolithic counters and SHA-1 authentication. As before, we note that split counters by themselves may seem a marginal improvement and that most of the benefit is due to the GCM authentication. However, we note that our split counters nearly halve the IPC overhead, from 8% in *Mono + GCM* to only 5% in *Split + GCM*.

We repeat the experiments from Figure 12 with different authentication requirements,

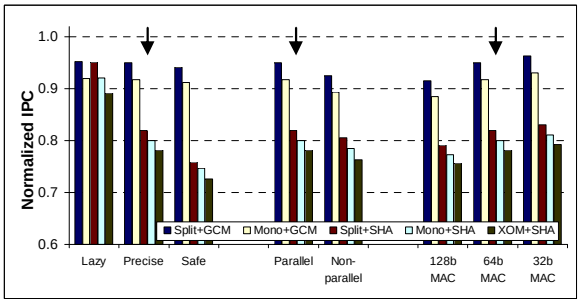


Figure 13: The average IPC comparison for different memory encryption and authentication scheme combinations with different authentication requirements, fetching schemes, and size of authentication codes.

with and without parallel authentication of Merkle tree levels, and different authentication code sizes. The default configuration is, again, indicated by an arrow in each set of experiments, and only the parameter indicated is changed while others remain at their default values. These results confirm our previous separate findings for GCM and split-counter mode, and indicate that our new combined scheme consistently outperform previous schemes over a wide range of parameters, and that each of the two components of the new scheme (split-counter mode and GCM) also consistently provides performance benefits.

3.8 Discussion

Protection from hardware attacks such as snoopers and mod chips has been receiving increasing attention in computer architecture. In this chapter we present a new combined memory encryption/authentication scheme. Our new split counters for counter-mode encryption simultaneously eliminate counter overflow problems and reduce per-block counter size to improve their on-chip caching. We also dramatically improve authentication performance and security by using GCM authentication, which leverages counter-mode encryption to reduce authentication latency and overlap it with memory accesses. Finally, we point out that counter integrity should be protected to ensure data secrecy.

Our results indicate that our encryption scheme has a negligible overhead even with a small (32KB) counter cache and using only eight counter bits per data block. The combined encryption/authentication scheme has an IPC overhead of 5% on average across SPEC CPU 2000 benchmarks, which is a significant improvement over the 20% overhead

of existing encryption/authentication schemes. Our sensitivity analysis confirms that our scheme maintains a considerable advantage over prior schemes under a wide range of system parameters and security requirements.

Security support in processors is a much desired feature nowadays. Secure processor can easily find its role in many occasions. For example, corporate users can rest assured with all their computers equipped with secure processors to protect against bus snooper attacks. Software and media distributors can protect their revenues with the help from secure processors to guarantee that end users consoles can only run with authenticate copies. Device manufacturers can shift critical security tasks to secure processors which help simplify the system design and provide a more secure boundary to attacks.

However, what blocks the widely adoption of secure processor in today's world is the significant performance overhead that accompanies the secure feature. Software developers and device designers strive really hard to squeeze every bit out of modern processors to achieve high performance. A performance degradation of 2× slowdown in worst case application with existing secure architecture is simply unacceptable. End user may not notice the performance impact of using secure architecture when the running application does not have much processor-memory communication. However, a 2× slowdown in memory intensive applications is hard to neglect. Such a worst case experience can easily kill the adoption of secure architecture for modern processors.

Our work dramatically closes the gap between the rosy prospect of secure processor and the bitter reality it faces today. By using split counters, it achieves satisfactory counter cache hit rate without a much increased on-chip storage budget. By overlapping memory authentication latency with memory access latency, it makes the protection for information integrity affordable. On average, our proposed scheme records a 1.05× slowdown across all evaluated benchmark applications. In contrast, the existing secure architecture introduces a 1.25× slowdown for the same suite of benchmark applications. Even for the worst case application, our proposed scheme records a 1.25× slowdown which beats the 2× slowdown in existing secure architecture easily.

Other than the performance benefit brought by our proposed scheme, the small storage

overhead contributes as well to promote the adoption of secure architecture we proposed. As modern processors become more and more versatile in functions and take over more and more tasks in systems, security support is not the only candidate which requires extra storage budget from the processor design. Though the importance of security support is widely recognized, secure architecture still often yields its way to performance related features when the tight storage budget disallow the coexistence of the two. The reason behind this is that the differences among processors in terms of security support cannot be reflected in existing processor benchmarks. As a result, it is far more difficult to market secure processors than those without secure feature but excelling in performance numbers. Our proposed scheme helps reduce storage overhead caused by applying secure architecture.

With our proposed scheme, we believe secure architecture becomes more affordable and easier to adopt by the industry.

CHAPTER IV

EXTENDING INTEGRITY AND CONFIDENTIALITY PROTECTION TO DISTRIBUTED SHARED MEMORY MULTIPROCESSORS

4.1 Motivation

In a multiprocessor system, data can be moved off-chip when it is sent to the main memory (like in uniprocessors), or when it is being transferred to another processor chip. Multiprocessor systems can be broadly classified into two groups, Symmetric Multiprocessors (SMPs) and Distributed Shared Memory multiprocessors (DSMs). In an SMP, a single broadcast communication medium (e.g. a bus) is used for processor-to-processor communication and for access to the centralized system memory. In a DSM, the communication medium is typically a point-to-point network which connects processor *nodes*, and the system memory is distributed among the nodes.

Architectural security support was proposed in prior work for both the SMP systems and the DSM systems. For SMP systems, prior work proposed to use the shared bus for all processors to keep track of the same encryption or authentication information. However, because the shared bus limits scalability of SMP machines, larger-scale machines typically use a DSM design where a shared communication medium that all processors can monitor is not available. In prior work for providing secure model for DSM systems, the problem of encrypting the memory is divided into two sub-problems. The scheme differentiates between processor-to-memory communication and processor-to-processor communication across the interconnect, and protects each with a separate security mechanism. Previously proposed techniques for uniprocessors are used to encrypt data communicated between a processor and its local memory. To encrypt data sent during processor-to-processor communication, per-processor-pair counters are kept to encrypt and decrypt data sent from one processor to another. When one processor needs to send data to another, the pad shared between the two is used to encrypt and decrypt the data. After the transaction, each processor increments the

shared counter and pre-computes the next pad. Because inter-node communication involves two separate security mechanisms, we refer to this approach as a *two-level* approach.

Such a two-level approach involves a large number of cryptographic operations for remote requests, and it results in several inefficiencies. First, the latency-hiding techniques associated with the cryptographic operations must *simultaneously succeed* in order to fully hide the total latency of the cryptographic operations. Since in reality this is difficult to achieve, the two-level approach frequently exposes some cryptographic latencies and exhibits high execution time overheads. Secondly, a large number of cryptographic operations cause heavy utilization of the hardware cryptographic engines, which may increase cryptographic latencies due to *contention*. Finally, authentication-related operations (e.g. MAC/signature generations and verifications) are directly in the *critical path* of a remote request for a data block because the authentication of one mechanism must be completed before passing a data block to the next mechanism. For example, on a remote read, the data block must be first authenticated by the home node using the processor-memory scheme before it is safely sent to the remote requestor using the processor-processor authentication mechanism.

Overall, a two-level approach is inherently performance inefficient. However on first examination of the problem of DSM protection, it is clear why such a two-level scheme might be chosen. Fundamentally, processor-processor communication seems best suited to a communication based protection scheme (i.e. associating a MAC value with each message), while processor-memory communication seems best suited to a storage based protection scheme (i.e. a Merkle tree covering the memory). However, to overcome the performance inefficiency introduced in the two-level approach, we need a single, unified protection scheme that is able to handle all types of data communication in a DSM system efficiently.

4.2 Overview

In the previous chapter, we discuss secure processor designs for single-chip uni-processor Systems and the related prior work in this category [9, 10, 26, 27, 28, 43, 44, 45, 50, 53, 55, 56]. Secure processor designs for bus-based Symmetric Multiprocessors (SMPs) have also been

proposed in [6, 43, 56]. However, secure Distributed Shared Memory (DSM) multiprocessor system design has not been thoroughly investigated, with still significant performance and storage overhead in previous proposed schemes [41, 25]. This is unfortunate because multiprocessor computer systems are currently widely used in commercial settings. These multiprocessors often run applications that operate on sensitive data, such as customer records, credit cards numbers, financial information, etc. Obtaining or modifying such data can be very lucrative to attackers, so this data should be protected from unauthorized access. Also, secure DSMs are increasingly needed as *utility* or *on-demand* computing proliferates. In utility computing, companies "lease" computation and storage resources of a large-scale, powerful system (e.g. the HP Superdome [34]) to customers who need such resources on a temporary basis or who want to offload their IT operations. These large-scale DSM systems are not under the control of the customers who are using their resources, so naturally customers often demand that the secrecy and integrity of their data be ensured. Indeed, industry analysts have reported that security concerns have partly restricted adoption of the utility computing model [2].

Certainly, utility computing providers would take steps to avoid unwanted physical tampering to their DSM system through enforcing physical security, such as by restricting physical access to the DSM to a few "trusted" employees and contractors. However, one of the key principles of security is that relying only on one layer of security is *risky* and is often *insufficient*. The risk of security attacks by select employees or parties that are trusted with physical access to the machine should not be underestimated. We can take an example from the case of Automated Teller Machines (ATMs) which also employ a certain level of physical security. While this physical security may deter common types of attacks, the level of security is often insufficient as evident in the report by Global ATM Security Alliance (GASA) that states that more than 80% of computer-based bank-related frauds involve employees [24]. In the case of DSM systems used for utility computing, the large amounts of sensitive data in these systems create a financial incentive for the attackers to perform corporate espionage or other malicious intents.

Additionally, physical or hardware attacks on DSM systems may be performed more

easily than on uniprocessor systems. For example, to snoop processor-to-memory communication in a uniprocessor system, attackers must tamper with the motherboard of the system. However, in a DSM system processor-to-processor communication is also vulnerable. Interconnect wires are exposed at the back of the server racks, and snooping devices that can store gigabytes of data (similar in principle to a *keyboard logger*) can be inserted without much disruption to the system. This *lack of disruption* is an important criteria for an attacker since an attack can be performed quickly and without leaving traces.

The possibility of hardware attacks may prompt customers to demand that DSM utility computing systems be equipped with secure hardware features that make them resistant even to hardware attacks. Utility computing providers that offer these features have an important competitive advantage compared to those who do not. Hence, we believe that data security in DSM systems will become an increasingly important issue in the future.

In this chapter we propose a new and *efficient* memory encryption and authentication solution for protecting the confidentiality and integrity of data in a DSM system. Our solution removes the inefficiency of the two-level memory encryption and authentication by using a single mechanism for both processor-to-memory and processor-to-processor communication. We refer to our approach as *single-level* memory encryption and authentication. The efficiency of our single-level approach comes from a significant reduction of cryptographic operations on remote requests. For example, when a remote requestor asks for a data block from a home node, the home node simply forwards the data block to the requestor (without authenticating, decrypting and re-encrypting it). Only the requestor needs to perform the cryptographic operation to decrypt the data block. This not only reduces the amount of cryptographic work involved, but also reduces the possibility of contention-related latencies at the cryptographic engines. In addition, only one latency-hiding mechanism is involved, which has a higher chance of succeeding compared to the simultaneous successes of multiple latency-hiding mechanisms in the two-level approach. Finally, using a single-level approach avoids transitions between two security mechanisms, allowing authentication of a data block on a remote request to be moved *off the critical path*.

Overall, we find that our techniques can provide secure data encryption and authentication in a DSM system with very low overheads. Passive or eavesdropping attacks are avoided by encrypting off-chip data communication, while active attacks involving message modification, injection, and deletion, are detected as failed authentication or coherence protocol errors. Our simulation results across all SPLASH-2 [52] benchmarks show that the average overhead of our mechanisms on a 16-processor DSM system is less than 1.6% with a maximum of 7%, relative to an identical system but without any memory encryption and authentication support. Compared to a two-level approach which has an average overhead of 5.3% and worst-case overhead of 16.3%, our single-level approach shows a reduction in overheads by a factor of 3.3× on average, and by a factor of 2.3× in the case with least improvement. This is important since DSMs are typically very expensive and run performance-critical applications. Overheads as large as those seen in certain applications under a two-level scheme are likely to be intolerable. Our single-level scheme reduces the overheads of these problem applications significantly, to a much more acceptable level. Additionally, the overheads under our single-level scheme are much more stable than those under a two-level scheme, which would give customers confidence that their applications would perform well. We also show that the low overheads of our technique hold across different number of processors in the DSM and different L2 cache sizes.

The remainder of this chapter is organized as follows. We discuss our assumptions and attack model in Section 4.3. Section 4.4 introduces our single-level memory encryption scheme for DSM systems. Section 4.5 details our evaluation setup. Section 4.6 presents our evaluation results and insights. Finally, we conclude with Section 4.7.

4.3 Attack Model and Assumptions

Before we present our solution for ensuring data confidentiality and integrity in a DSM system (Section 4.4), we discuss the types of attacks that we assume possible in a DSM system. Our attack model is the same as one assumed in the study by Rogers et al. [41], and we will briefly overview it in this section.

Like other work on secure processors [6, 9, 10, 14, 26, 27, 28, 41, 43, 44, 45, 50, 53, 55, 56],

our first assumption is that data located on a processor chip is secure, while data located off-chip is vulnerable to attack. In a DSM multiprocessor, the main off-chip structures that are vulnerable to attack are the interconnection network which connects the processors, and the local main memory of each processor (memory bus and DRAM). Such an assumption is also used in commercial secure processors such as IBM SecureBlue [14] and Dallas Semiconductor DS5002FP [30].

In addition, we assume that attackers can target major off-chip structures such as the local memory bus and DRAM, and the interconnection network. The types of attacks that may be attempted on these structures include *passive* attacks (eavesdropping) and *active* attacks (altering data in memory or messages on the interconnect and local buses). Through active attacks, attackers may modify headers or the data payload of a coherence message, as well as replay an old version of a message. We refer to the latter attack as a *message replay* attack. Attackers can also act as a *man-in-the-middle*, dropping and injecting messages in the interconnect.

We assume that attackers are interested in discovering secret data, and hence our goal in protecting against passive attacks is to always ensure privacy through encrypting all data communicated off-chip between processors and their local memories or between different processors.

We assume that *traces of attacks* which we define as detectable anomalous behavior, such as cryptographic errors (failed authentication), invalid coherence messages, or system crashes, are sufficient deterrents to active attacks. The rationale for this assumption is that since DSM systems are typically large and expensive, they are likely protected with reasonable physical security that includes restricted access to a few select employees or contractors. Therefore, attackers are likely malicious employees or contractors with access to the system. We have discussed in Section 4.2 that this is a real possibility as hinted by a case study in Automated Teller Machines [24]. Such attackers will likely prefer performing an active attack that produces no *traces* that can alert other users and allow an investigation to correlate the traces to the attackers. Consequently, it is sufficient for our security mechanism to ensure that all active attacks produce traces that are detectable. We also assume that a

secure mechanism is in place to log these traces and alert users when a log is created.

We also assume that *precise authentication*, which refers to delaying load instruction retirement (or even load data use) until data authentication is completed [43, 53], is not needed. The rationale for this assumption is that a hardware attack takes time to perform, so detection within a reasonable time window (say, thousands to millions of cycles) serves as a sufficiently powerful deterrent to attackers. In our design, detection of an attack occurs within the latency of a message round trip, which is on the order of hundreds to thousands of cycles.

A secure DSM system with memory encryption support must have mechanisms in place to provide secure booting of the machine and secure key setup. This is an important issue that is beyond the scope of this study. In this study, we simply assume that there is a mechanism in place to guarantee secure booting and key set up on the machine.

Finally, we do not assume any specific configuration of the interconnect network such as message delivery ordering or routing strategies. However, we assume that the DSM employs a home-based coherence write invalidate protocol, and we specifically illustrate our algorithm with the MESI protocol in mind.

4.4 Data Protection for DSM

In this section we present our new techniques for fast and secure data encryption and authentication in DSM systems. We begin with an overview of the cryptographic operations used in our scheme (Section 4.4.1) and a brief discussion of the challenges in providing secure DSMs (Section 4.4.2), followed by a discussion of how we perform data encryption (Section 4.4.3) and authentication (Section 4.4.4) in a DSM system. We close the section with a security analysis of our scheme (Section 4.4.5).

4.4.1 Overview of Cryptographic Operations

As we illustrated in chapter 2, popular cryptographic functions such as 3DES/AES encryption and SHA-1/MD-5 authentication can be used in a direct encryption or *electronic codebook* (ECB) *mode*, where these functions are applied directly to the plaintext of the

data to produce its ciphertext and a MAC (Message Authentication Code). One drawback of ECB in the context of memory encryption is that decryption and authentication of the ciphertext cannot begin until it is fetched on-chip. Consequently, cryptographic latencies are added directly to the critical path of off-chip data fetches. Another drawback is a security concern in that the statistical distribution of ciphertexts matches that of the plaintexts. Therefore, in this study, we focus our solution around counter-mode encryption and authentication.

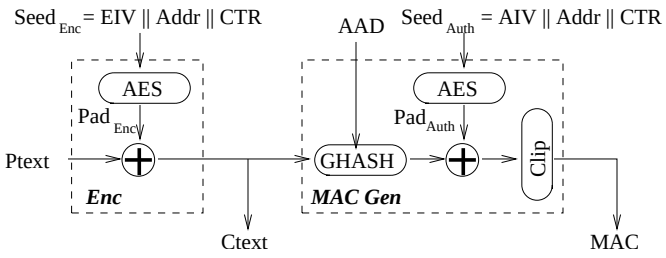


Figure 14: Galois/Counter Mode Encryption and MAC Generation used in our scheme.

Counter-mode encryption assigns a counter for each data block that is used for encrypting/decrypting the block, and for generating a MAC for the block. While there are various counter mode encryption and authentication algorithms, Galois/Counter Mode (GCM) [54, 31] we discussed in pervious chapter is one of the most promising ones. GCM combines encryption and authentication operations very efficiently, has been demonstrated to be as secure as the underlying AES encryption algorithm [54, 31], and has been illustrated in chapter 3 to be effective in providing architectural support for data security in uniprocessor environments [41, 53]. Figure 14 illustrates GCM with encryption in the left dashed box and MAC generation in the right box. In encryption, a unique *encryption seed* is input to the AES encryption unit to obtain an *encryption pad*, which is XORed with the plaintext to obtain the ciphertext. To guarantee security, a *necessary* condition in counter-mode encryption is that each encryption pad value can only be used for encryption once, so consequently the encryption seed value must also be used only once. To encrypt a cache block, typically the encryption seed can be composed of the concatenation of *block address* (to ensure spatial uniqueness), a *per-block counter* that is incremented after each use (to ensure temporal uniqueness), and an *encryption initial vector* (EIV – to ensure that pads

used for encryption are different than ones for other uses).

To generate a MAC for a block, an *authentication seed* is input to the AES encryption unit to obtain an *authentication pad*. The authentication seed must also be unique for each use, so similar to encryption seed, it can be composed by concatenating an *authentication initial vector* (AIV), block address, and a per-block counter. The ciphertext together with *additionally authenticated data* (AAD) are input to a GHASH function. The AAD contains data that needs to be authenticated but not encrypted. The GHASH function consists of a short chain of bitwise XORs and Galois Field multiplications. Finally, we can clip the MAC to the desired length based on security requirements by taking its most significant bits (MSBs). It is worth noting that a MAC in GCM is generated using a secret key. Consequently, given a ciphertext, an attacker cannot generate a valid MAC for it.

Performance-wise, it is important to note that most of the cryptographic latency comes from generating encryption and authentication pads. Once these pads are available, a ciphertext can be obtained through a bitwise XOR, and a MAC can be obtained through the GHASH and XOR operations. All operations past the pad generation only require a few cycles to complete. As a result, we primarily need to focus on hiding the latency of pad generation.

4.4.2 Challenges in Providing Secure DSM Architectures

Secure processor architectures, which include counter-mode encryption and Merkle Tree authentication, already have the capability of providing secrecy and integrity to data in the local memory. However, data transmitted between processors are not automatically protected from attacks. The main challenges to providing such protection are: (1) how to allow data encrypted by one processor to be decrypted by another processor, and (2) how to protect data that is communicated among different processors.

The first problem, in counter-mode encryption, essentially implies that communicating processors must share the *same* counter that is used for encrypting a data block. However, counter sharing necessitates communicating its value across multiple processors, implying

that counters will be subjected to alteration attempts by attackers. In counter-mode encryption, it has been discussed in details in pervious chapter that if counter modifications are not detected, they can be used by attackers to break the encryption, for example by forcing reuse of old counter values (and thus reuse of encryption pads). Hence, to provide data encryption for multiprocessor systems, we have to deal with an open problem of how to provide *secure counter communication*. In the past, researchers avoided the problem of protecting counter communication by relying on information that is naturally shared by multiple processors without needing communication. In bus-based Symmetric Multi-Processor (SMP) system, the shared bus naturally provides a shared medium for all processors to use for processor-processor encryption. For example, a global bus counter can be used to encrypt data transfer between processors [43], or data transmitted on the bus itself can be used to encrypt new data blocks through Cipher Block Chaining [56]. Note that because the global bus counter or bus data is used for encrypting processor-processor communication (rather than the per-block counters used for processor-memory encryption), such an approach relies on two separate security mechanisms: one to handle processor-processor communication, and another to handle processor-memory communication. We refer to this approach as a *two-level* memory encryption and authentication.

The same principle of two-level memory encryption and authentication to avoid secure counter communication is applied in Distributed Shared Memory (DSM) multiprocessors by Rogers et al. [41]. Since DSMs use a point-to-point interconnect and lack a shared communication medium that all processors can monitor, Rogers' scheme relies on maintaining shared communication counters between each processor pair that are incremented by each processor independently after encrypting or decrypting a block communicated between the two. This way counters do not need to be communicated between processors in the normal case. In a two-level approach, since a remote read request involves two security mechanisms, there are two major performance drawbacks. First, in order for the full cryptographic latencies of a remote read request to be hidden, all latency-hiding techniques associated with each cryptographic operation must simultaneously succeed. This is illustrated in Figure 15(a).

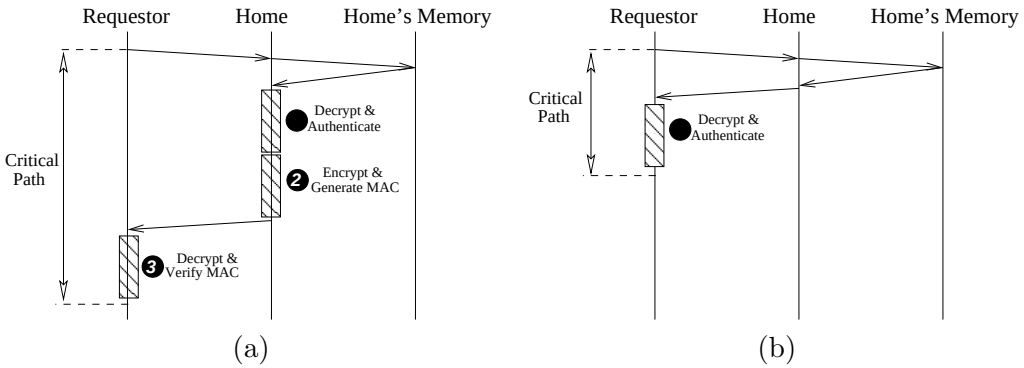


Figure 15: Comparing the Critical Path of a remote data request for Two-Level encryption scheme (a) and Single-Level encryption scheme (b).

The critical path of the two-level approach includes decryption and Merkle Tree authentication after local memory fetch (Circle 1), re-encryption of the requested block and its MAC generation using the processor-processor mechanism (Circle 2), and finally decryption and MAC verification at the requestor side (Circle 3). Each of these cryptographic operations incurs a latency that needs to be hidden by latency-hiding techniques such as counter caching, counter prediction, and pad pre-generation. Each technique is imperfect (the counter cache may miss, counter prediction may mis-predict, while pads may be pre-generated too late), and this contributes to the inability to fully hide cryptographic latencies. For example, if the success rate of each individual technique ranges from 70% to 80%, full latency-hiding only occurs between 34% to 51% of the time (i.e. because $0.7^3 = 0.343$ and $0.8^3 = 0.512$), which means that a significant fraction of the time only parts of cryptographic latencies are hidden. Note that this is in addition to non-hidable latency such as the GHASH function in GCM. Another major drawback is the large amount of cryptographic work which increases occupancy and contention at the cryptographic engine, which could further exacerbate cryptographic latencies. Overall, a two-level approach is bound to suffer from relatively high execution time overheads due to employing two security mechanisms.

We note that the drawbacks of the two-level approach can be avoided if we employ a unified encryption and MAC generation scheme for both processor-memory and processor-processor communication, as illustrated in Figure 15(b). On a remote request, the home node fetches the ciphertext of a data block, and immediately sends it off to the requestor.

Only the requestor performs cryptographic operations to decrypt the block and verify its MAC. This cuts down the total cryptographic latencies in the critical path by a *factor of three*. More importantly, because only one latency-hiding mechanism needs to succeed in order to hide the full cryptographic latency, a single-level approach achieves much lower and more stable execution time overheads than a two-level approach, as we will show in Section 4.6.

4.4.3 Single Level Memory Encryption

Now that we have established the benefits of a single-level approach compared to a two-level approach, we need to design mechanisms that can accomplish single-level memory encryption and authentication. In a single-level encryption scheme, when a requestor receives encrypted data, it needs to decrypt it using the counter value that was used to encrypt it. This per-block encryption counter was used to encrypt the data block in one node (possibly the home), and is needed to decrypt it in another node (the requestor). Logically, one may think that this counter can be cached at both nodes and can automatically be kept coherent by the coherence protocol. However, this would be a poor solution because a block's counter is incremented whenever the modified block leaves a processor chip (as a response to a remote request or on a write-back). At that time all other copies of the counter are invalidated. When another processor requests the block, the counter value it had cached has been invalidated and the new value must be fetched before the pads to decrypt and authenticate the data can be computed. As a result, the requestor would be unable to hide the decryption latency. Another serious problem with keeping counters coherent is that these coherence messages are themselves subject to replay attacks, so such messages must be authenticated, and this GCM authentication uses another counter, etc.

To avoid the complexity of relying on the cache coherence protocol to keep counters securely coherent, we adopt an alternative approach in which a *counter of a block can only be cached at the home node and at the owner node (if any).* That is, for a data block in clean state, only the home can cache the counter value. For a data block in the modified state cached at an owner, the owner eventually needs to encrypt the block and send it off to

another processor (due to intervention or write-back), and hence it is allowed to cache the counter value temporarily until the block is replaced or becomes clean. When a processor asks for an exclusive state for a data block through an upgrade or read-exclusive request, the home processor increments the counter value of the block, and replies with the new counter value to the requestor. Certainly, this counter value must be protected from tampering, so we protect it using the same mechanism used to protect data communication (to be discussed in Section 4.4.4). We will now discuss the latency-hiding aspect of this approach, and leave the security aspect until later in Section 4.4.4.

Hiding Decryption Latency at the Requestor. In our single-level memory encryption, typically the only node that performs a cryptographic operation is the requestor of a data block, which must decrypt the block before it can use it. The key to successful latency-hiding at the requestor is that the requestor must have the counter to pre-generate the appropriate decryption pad before the data arrives. Since the requestor is not allowed to cache the counter, the home node that caches the counter must supply the counter early. Fortunately, with simple coherence protocol modifications, this early supply of counter value is achievable. Figure 16 shows how various scenarios are handled. The first scenario is when the requested data is in the home node's local memory (Figure 16(a)). In this case, the home looks up its local counter cache to obtain the block's counter (CTR). If it finds the counter, it forwards the counter immediately to the requestor, and in parallel begins to fetch the data block ciphertext (CTEXT) from its local memory. The counter value would arrive at the requestor one memory-latency before the data, allowing the requestor to generate the decryption pad ahead of receiving the data ciphertext.

Figure 16(b) shows another scenario in which the data block requested is also in the home processor's cache (in plaintext form because it is on chip). Before forwarding the data block to the requestor, the home processor must first encrypt the data block. In parallel, it sends the counter value to the requestor. The requestor, upon receiving the counter immediately pre-generate the pad needed to decrypt the ciphertext of the data block that is still yet to arrive. In this case, the one pad generation latency at the home node is exposed, but the pad generation latency at the requestor is hidden. Again, this

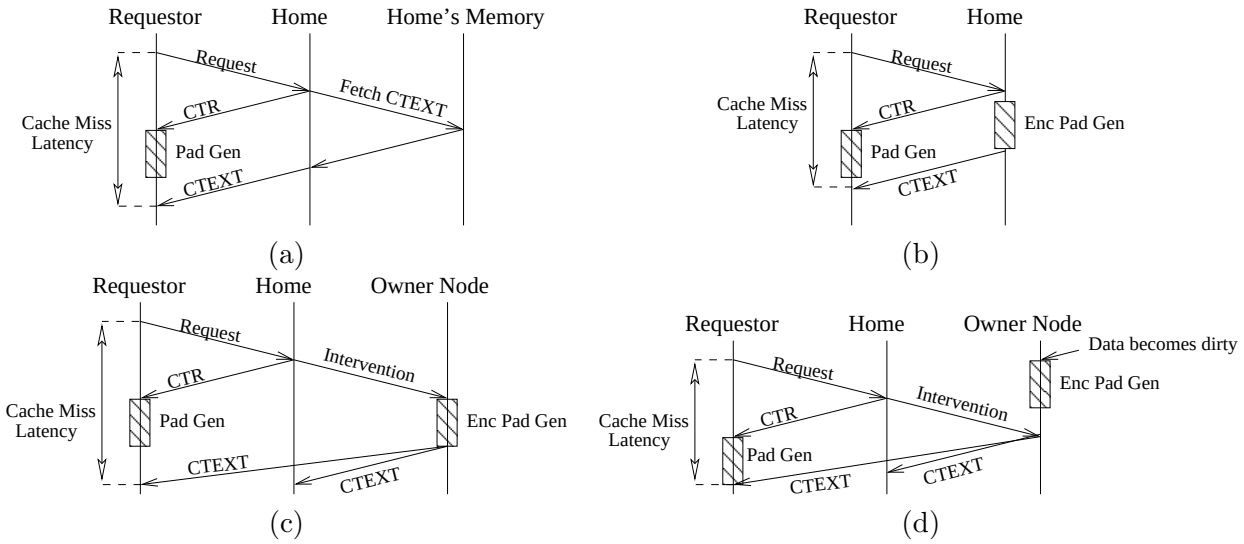


Figure 16: Coherence protocol modifications for hiding cryptographic latencies of remote data request in our single-level counter mode memory encryption when data is supplied by the home node's local memory (a), home node's cache (b), remote owner (c), and remote owner employing pad pre-generation using owned block pad buffer (d).

is still better than the two-level approach which in the worst case can suffer up to three pad generation latencies. Furthermore, the scenario in Figure 16(b) rarely occurs because typically different processors work on different data, so a remote read rarely finds the data in the *clean* state at the home processor's cache (if the state of the data is dirty, it is handled in the next scenario).

Figure 16(c) shows a scenario in which the data is owned by another processor. In this case, the home node will forward the request to the owner, and in parallel send the counter value to the requestor. When the data block owner receives the request, it will first encrypt the data block and send it to both the home processor and the requestor. The requestor still receives the counter before it receives the data, and can overlap the pad generation with the communication latency. However, one pad generation latency at the owner is exposed. Unfortunately, compared to the previous scenario in Figure 16(b), this case quite frequently occurs when different processors share data that is frequently written (both true and false sharing). Hence, we seek ways to optimize this scenario next.

Hiding Encryption Latency at the Sender/Owner. Note that in Figure 16(c) the encryption pad generation at the owner's side is on the critical path, so we can optimize

this scenario further (illustrated in Figure 16(d)). To achieve that, when a data block becomes dirty or modified, immediately we trigger its encryption (and authentication) pad generation. These pads are then stored in a small buffer called the *Owned-Block Pad Buffer*. We only pre-generate pads for modified blocks because these are the only blocks that have a possibility to be intervened by another processor. When an owner receives an intervention for a data block, if it finds pads for the block in the owned-block pad buffer, it can immediately encrypt and generate the MAC for the block in a few cycles, and send the block off to the requestor.

One important question would be how large the owned-block pad buffer needs to be. We observe that frequently communicated blocks typically reside in any one cache for only a short time since they receive interventions quite frequently (e.g. shared locks). We also observe that data blocks that are in the modified state but have not been intervened for a long time are unlikely to be intervened frequently. Therefore, the owned-block pad buffer only needs to store pads for the blocks which are most recently upgraded to a modified state. For a block that is not intervened, its pads will naturally reach the LRU entry in the owned-block pad buffer and be replaced. We find that a 32-entry owned-block pad buffer is sufficiently large to ensure that in most cases, pre-generated pads are available for intervened data blocks. Since pad pre-generation is potentially useless (for blocks that are not intervened), to conserve AES engine bandwidth, we perform pad pre-generation only when the AES unit is idle.

Local Counter Management and Counter Prediction. Note that Figure 16 assumes that the home finds the counter in its local counter cache for each request. If this is not the case, the home must first fetch the counter from the local memory before the counter can be sent to the requestor. The frequency of this case can be made very rare with a good counter caching policy at the home. Note that a home node only caches counters of blocks that are in its local memory, hence the counter cache size can be kept constant regardless of the number of processors in the DSM. In fact, when a cache block is shared by multiple processors, the first processor that requests the data block from the block home node will cause the home node to cache the counter of the block, effectively prefetching the

counter for subsequent processors that want to fetch the block. In addition, since counters are typically small (e.g. 8 bits), a counter cache block can hold many counters from many cache blocks. This results in good spatial locality in that bringing a counter block to the processor chip helps *prefetch* counters of other neighboring data blocks. Hence, the common case is that the home finds a counter of a requested block in its local counter cache.

Furthermore, the requestor can employ a *counter prediction* mechanism such that even when a home node cannot find the counter in its local counter cache when there is a request for the block, the requestor can still pre-generate pads ahead of time. For example, each processor can keep track of the last counter value of remote blocks that it has recently evicted. If the last value is zero (indicating a read-only data block), we find that there is a very high probability that the counter value is still zero, hence we can start pad generation assuming that the counter value is zero. In addition, tracking such zero-valued counters is very space efficient since we only need a single bit per counter. We use a *mask buffer* to store these bits, with a bit value of one indicating a particular counter was last seen as zero. We find that a 32 entry mask buffer is able to provide effective counter prediction of zero-valued counters. When a node sends a data request to a remote node, it can send along the predicted counter value that it used for pregenerating its pad. If the predicted value of the counter is correct, the home node can skip sending the counter value to the requestor, hence we have an additional benefit of conserving network bandwidth. We note that there are many other ways for predicting counter values and it is beyond the scope of this study to search for the best counter prediction scheme exhaustively. However, we find that even the simple counter prediction scheme we just discussed can achieve high accuracy of 76% (Section 4.6).

Summary. Overall, we have shown that cryptographic latency-hiding can be achieved through simple coherence protocol modifications that allow the home node to forward counter values, pad pre-generation at owner nodes for modified blocks that they cache, and relatively simple counter prediction at the requestor. With these mechanisms in place, pad generation latency is exposed in only a few cases, and even in these cases, only one pad generation latency is exposed. This compares favorably with the two-level approach in

which up to three pad generation latencies may be exposed on a remote request.

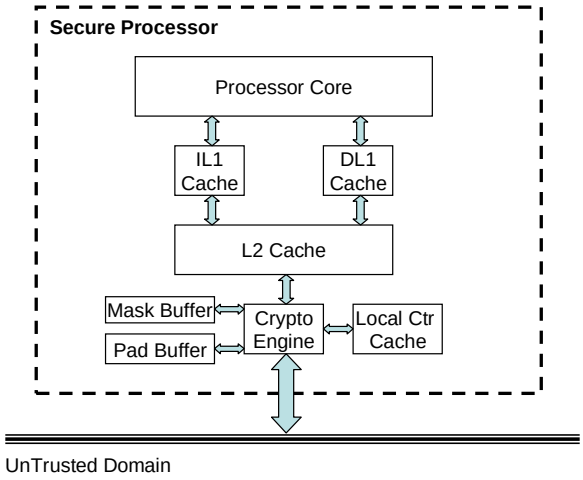


Figure 17: Architecture modifications to a node

The architecture components that we add to each processor are minor as shown in Figure 17. A secure processor architecture for uniprocessor systems already contains a cryptographic engine and local counter cache. Over them, a 32-entry owned-block pad buffer to hold pre-generated encryption and authentication pads, and a 32-entry mask buffer to store the bit masks indication zero-valued counters for counter prediction.

4.4.4 Single-Level Memory Authentication

So far we have discussed how cryptographic latencies of our single-level memory encryption and authentication can be hidden with simple modifications to the coherence protocol, and the use of a counter cache and owned-block pad buffer. In this section, we will discuss the security aspect of our scheme. First, passive attacks are already protected against in the previous section, since data is always communicated as ciphertext. So in this section we discuss mechanisms to ensure that active attacks are detected.

Memory authentication in DSM systems requires protection not only from attacks against data loaded from a processor's local memory, but also from attacks against data communicated between processors through messages across the system interconnect. Complete protection of data loaded from off-chip memory has been proposed in a prior study [9] in the form of Merkle tree authentication for uniprocessor systems. As with encryption, it is not an easy task to extend the Merkle tree scheme to authenticate both data loaded from

local memory and data received from a remote processor. It may seem possible to use a single Merkle tree to cover all of the DSM memory, and to distribute this Merkle tree across the processors, keeping the tree nodes coherent using the coherence protocol. However, this solution has similar problems to our discussion in Section 4.4.3 of trying to distribute per-block counters and maintain their coherency for memory encryption. For example, if we use one Merkle tree for the entire memory, when the tree root is modified by one processor it must be communicated to others. Since the integrity of the entire Merkle tree depends on the root being completely secure on-chip, these frequent messages that transport the root across the network are a major liability. Also, data updates result in modifications of stored Merkle tree nodes, which results in invalidations of these nodes in other processors. In particular, when a requesting processor receives a data block, the Merkle tree node needed to authenticate the block has just been invalidated because the block has previously been modified by its owner.

To avoid the tremendous complexity of supporting a coherent global Merkle Tree, we adopt an alternative in which each node maintains its own *local Merkle Tree* that covers only the node's local memory. In addition to using a local Merkle tree to protect a node's local memory, we use a separate lightweight message authentication protocol to provide authentication for all data communicated in a DSM system.

In a DSM system, in general, a data block may be transferred between processors in three cases. The first is when a processor requests data, and the home node replies with the data. The second case is when a processor requests data, and another processor that owns the block in a modified state supplies the data. The final case is when the current owner of a data block needs to flush or writeback the block to its home node. We will now describe how the data is authenticated in each of these cases.

4.4.4.1 Authenticating Data sent to a Requestor by the Home Node

The first case we will discuss is when a processor requests data, and the home node of the requested data block replies with the data block. In Figure 18 we show the steps of our process to authenticate data communicated to a requesting processor. This scenario

matches that in Figure 16(a), except that now we augment its security through message authentication. When a requesting processor R sends a data request to the data's home node, the home node fetches not just the data ciphertext (CTEXT), but also its MAC – which is the lowest level MAC in its Local Merkle Tree generated as in Figure 14 (Circle 1). Then the home node sends off the data, its MAC, and a *message counter* (MCTR) to the requestor (Circle 2). Each processor keeps an on-chip message counter that it increments each time it sends a coherence message to another processor. The purpose of this counter is to detect message replay attacks and it will be elaborated on later. When the requestor receives the data, it decrypts the data using the pad that it generated earlier. Then it proceeds with verifying the MAC of the ciphertext, which verifies that the ciphertext, counter, and MAC combination are valid (Circle 3). Attackers cannot modify the ciphertext and/or MAC and/or the counter with arbitrary values without incurring a MAC mismatch [1]. In addition, since the MAC is generated using a secret key, given a ciphertext, attackers cannot produce a valid MAC. The only thing that attackers can do is to replace the ciphertext, counter, and MAC to a *known valid combination*, such as taken from other prior messages. To detect such tampering, we utilize another MAC that is computed over the received message. We refer to it as a *message MAC* (MSGMAC), and its computation is the next step performed by the requestor (Circle 4). MSGMAC is computed using the following equation:

$$MSGMAC \Leftarrow AES_K(MSGAIV||PID(H)||MCTR) \oplus GHASH(PID(H)||MCTR||MAC) \quad (1)$$

The part to the left of the XOR is the authentication pad for this new message, which is a basic component for counter-mode authentication. The requirement for an authentication pad is that its value cannot be reused, hence we need to ensure its uniqueness. To ensure uniqueness of message authentication pads, we form the authentication seed by concatenating a message authentication initial vector (MSGAIV) which is not the same as

[1] Recall in Figure 14 that a counter value is an input to the MAC generated using GCM authentication. As a result, different counter values generate different MAC values.

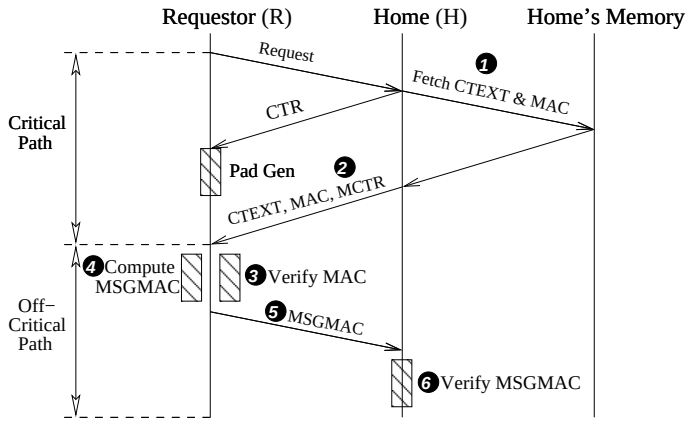


Figure 18: Steps for authenticating data sent to a requesting processor across the interconnect.

the initial vectors used for encryption or Merkle Tree MAC computations, the processor ID of the home node (PID(H)), and the message counter (MCTR). Since each home node has its own message counter, the concatenation of PID(H) and MCTR creates a unique combination in the system. The parts inside the GHASH function are the items that we want to authenticate. It contains PID(H) as well as MCTR and the MAC that were received. The MSGMAC is essentially a unique signature of PID(H), MCTR, and MAC. Attackers cannot modify any of PID(H), MCTR, MAC, or MSGMAC without causing a MSGMAC mismatch.

Next, the MSGMAC is sent back to the home node (Circle 5) which verifies it (Circle 6). The verification involves recomputing MSGMAC at the home node using the PID(H), MCTR, and MAC that the home node *knows* it sent to the requestor. If any of PID(H), MCTR, and MAC have been modified by attackers, the requestor would have sent a wrong MSGMAC and it will mismatch with the MSGMAC computed by the home node. The reason is that while the MAC verification at the requestor cannot detect a replay attack where the ciphertext, MAC, and message counter are changed to some known valid combination, the MSGMAC generated and sent by the requestor captures the essence that the MAC has changed. Thus, MSGMAC will mismatch at the home node. Attackers could not have generated a fake MSGMAC that matches the MSGMAC computed by the home node for two reasons: (1) they do not have the secret key to compute a valid MSGMAC, and (2) MSGMAC value is never repeated in the system because MCTR is always incremented

and hence each MSGMAC value is always new and could not have been seen before by the attackers, and (3) the home node always uses valid MCTR and MAC values to compute MSGMAC since they are kept on chip.

Note that one important feature of our single-level authentication is that the critical path of a remote read is unaffected by the authentication scheme. Indeed, authentication steps (Circle 3, 4, 5, and 6) all occur off the critical path and can be performed in background. This is in contrast to the two-level authentication which involves MAC generation and verifications in the critical path (Figure 15(a)).

4.4.4.2 Authenticating data written back to its home node

Our authentication steps for data communication caused by an owner node writing back a data block to the home node are shown in Figure 19. First, the current owner of the data block encrypts the data to generate the block ciphertext (CTEXT) and MAC using the current value of the block counter (CTR). Then, the owner sends a write back message containing CTEXT and MAC to the home node of the data block (Circle 1). Note that the block's counter CTR does not need to be sent since the home node always knows the current value of CTR. The home node then verifies the received MAC using the received CTEXT and its known value of CTR.

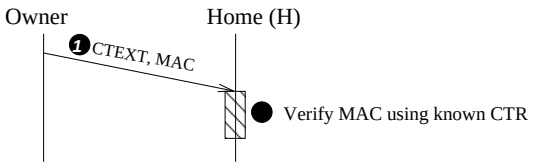


Figure 19: Steps for authenticating data written back to its home processor across the interconnect.

Attackers cannot alter one of the ciphertext or MAC without triggering failed MAC verification at the home node. In addition, attackers cannot alter the ciphertext and produce a valid MAC for it because they do not have the secret key necessary to compute a valid MAC. Finally, changing both the CTEXT and MAC pair to a known valid combination (such as in replay attack) will also result in failed MAC verification at the home node because one of the MAC's input's is the counter value of the block. Also, because the

counter value has not been used in the past, the valid MAC value has also not been seen in the past. Since the fresh counter value is always kept safe at the home node, it cannot be altered by attackers to subvert the MAC verification at the home node.

4.4.4.3 Authenticating Data sent to a Requestor by an Owner Node

The last case is when a requested data block does not reside in the home node because it is in a modified state at a remote owner. In this case, the remote owner needs to send the requested data block to the requestor, as well as write it back to the home node so that the resulting state is clean. The protocol to achieve security in this case is similar to Figure 18, except that in this case the sender is not the home node, but the owner node. So the owner node encrypts the requested data block and computes its MAC. All the needed inputs for encryption and MAC generation are on chip (the data plaintext, counter value, and message counter). Then it sends off the ciphertext, MAC, and its message counter to the requestor. After the requestor verifies the MAC and computes MSGMAC, the requestor sends the MSGMAC back to the owner which then verifies the MSGMAC.

However, there is one difference compared to the case in Figure 18. In this case, the owner needs to write back the data block to the home node so that the state of the block can be updated to clean. Hence, this case also uses the secure write-back authentication shown in Figure 19.

4.4.5 Security Analysis

Table 3 summarizes the types of attacks that may be attempted by attackers, how they can be detected by the system, and a short explanation for such detection. Since every data transfer over the interconnect is protected with a cryptographically secure MAC computed over the data ciphertext, the data address, and its per-block counter value, an attacker cannot tamper with or replace any data value without triggering MAC mismatch. In addition, all MAC transfer is also protected by the MSGMAC, so even if the data ciphertext, MAC, and MCTR values are changed to known valid combination, the MSGMAC will mismatch at the home node.

There are some cases that may not result in failed authentication, however as we can

Table 3: How different types of attacks are identified and explanation as to why they are identified.

Type of Active Attack	Reaction	Key Reasons
Alter data and/or MAC	MAC mismatch	MAC is recomputed at requestor
Alter data, MAC, and CTR	MSGMAC mismatch	MSGMAC is validated at the home/sender using known CTR and MAC values
Replay old data, MAC, and MCTR	MSGMAC mismatch	MSGMAC is validated at the home/sender. MCTR cannot be altered since it is known to the home/sender
Message dropped	Timeout or anomalous coherence protocol behavior	Data secrecy not violated
Fake request message injected	MSGMAC mismatch or Anomalous coherence protocol behavior	Home/Owner responds but gets invalid MSGMAC
Fake response message injected	MAC mismatch	Attackers do not have the key to generate a valid MAC
Fake MSGMAC message injected	MSGMAC mismatch	Attackers do not have the key to generate a valid MSGMAC
Alter message sender ID	MSGMAC mismatch	sender ID is an input to MSGMAC computation
Alter message receiver ID	Anomalous coherence protocol behavior	Messages received by processor which are not expecting it
Alter message type	Anomalous coherence protocol behavior	

see they will result in anomalous coherence protocol behavior or timeout. For example, if a message is dropped, the sending node will not receive an acknowledgment and will time-out, then it will try to resend the message. Or if a fake data request message is injected, the home or owner node may send off a reply containing the ciphertext, MAC, and message counter MCTR. The attacker still cannot break data privacy simply by looking the data ciphertext. If the attacker changes the header of a coherence message, such as changing the destination processor, or even alters the type of the message, this will result in anomalous coherence protocol behavior such as a node receiving a data block that it does not expect. Such behavior constitutes a trace that should be logged and alerts users since it may point out to instances in which attacks were carried out.

In summary, our proposed memory authentication scheme for DSM systems protects from all data tampering and replay attacks against data loaded from a local memory as well as data communicated between processors across the interconnect. In addition, our scheme reduces the amount of cryptographic work due to authentication that is on the critical path of fetching data from a remote memory compared to the two-level scheme proposed in [41].

4.5 Experimental Setup

We model our DSM system using a detailed execution-driven simulator based on SESC [15], an open source multiprocessor simulation environment. The simulated DSM system consists

of 16 2GHz, 3-way out-of-order issue processors as the default. Each processor has separate data and instruction L1 caches that are both 16KB, 2-way with 64B lines and a 2 cycle hit latency. The L2 cache of each processor has an 8-way associativity, 64B block size and a 10 cycle round-trip hit latency. The L2 cache size is varied from 128KB, 256KB, to 512KB, with the default of 256KB. The reason for choosing a relatively small L2 cache for the evaluation is to induce higher miss rates that would stress our single-level memory encryption and authentication more (refer to the Local L2 Miss Rate at Table 4). The memory system uses round-robin page allocation among the processors with 4KB pages. Each processor uses a 1 GHz, 4-Byte wide, split transaction memory bus to access the main memory with a 200 cycle uncontended round-trip latency. The processors are connected by a hypercube network with fixed-path routing. Each link has a bandwidth of 2 GB/s and the hop delay is 50ns modeled after the SGI Altix [46]. A MESI cache coherence protocol is implemented using a full bit vector with a home-based directory using reply forwarding.

The hardware for our protection scheme includes a default 32KB, 8-way counter cache to store the split-counters of frequently accessed blocks in a processor's local memory. This cache is the same as in a uniprocessor memory encryption scheme. For owned-block pad buffer, we use a 32-entry FIFO buffer having a total size of 1 KBytes. For counter prediction, we add a small 32-entry mask buffer for storing a bit vector of which data blocks last had a counter value of zero. The total size of the bit vector is approximately 512 Bytes because each entry is 16-byte in size. We assume a 2-cycle latency for accessing the pad buffer and mask buffer. The AES encryption engine is pipelined with an 80 cycle latency and 5 cycle occupancy. On a 2 GHz processor, this is comparable to the 37ns implementation shown in [19].

To evaluate our scheme, we use all 12 applications from the SPLASH-2 benchmark suite [52]. We use the standard input sets, and simulate all applications from start to completion with no fast forwarding or sampling. Table 4 shows the global and local L2 cache miss rates, and percentage of L2 miss rates satisfied by a home node's local memory, of each application running on the simulated DSM without any security protection.

Table 4: Global L2 Miss Rate, Local L2 Miss Rate, and percentage of L2 misses serviced by the home node of the data.

App	Glb L2 MR %	Local L2 MR %	% Home Reqs
Barnes	0.05%	3.3%	64%
Cholesky	0.24%	45.9%	88%
FFT	0.83%	63.0%	99%
FMM	0.04%	18.9%	60%
LU	0.05%	24.4%	80%
Ocean	1.05%	27.4%	75%
Radiosity	0.07%	28.4%	66%
Radix	0.65%	22.3%	96%
Raytrace	0.39%	22.3%	91%
Volrend	0.26%	34.6%	89%
Water-n2	0.04%	6.4%	50%
Water-sp	0.03%	12.1%	88%

4.6 Evaluation

4.6.1 Overhead of DSM Data Protection

In Figure 20 we show the execution time overhead our single-level DSM data protection scheme, normalized to a DSM system with no support for data encryption or authentication. For comparison, we also show the overhead of the two-level, CACHED scheme from the previous work on data protection for DSM systems [41]. We compare against this scheme since it is the only one which is similar to ours in that it meets the design criteria of small on-chip storage overheads and the ability to scale to arbitrarily large DSM system sizes (in terms of number of processors in the DSM). However, we note that this scheme on its own cannot prevent all data replay attacks, so we augmented it with the ability to detect replay attacks using our Message MAC technique discussed in Section 4.4.4. Thus, the two schemes are also comparable in terms of security strength.

From this figure, it is clear that while both schemes are similar in terms of small hardware support, the ability to support large systems, and security, our single-level scheme provides significantly better performance than the previously proposed CACHED two-level scheme. The average overhead across all applications of our scheme is 1.6% while the overhead of CACHED is 5.3%, representing a reduction of overheads by a factor of 3.3×. In addition, there are several applications which suffer from fairly significant overheads under

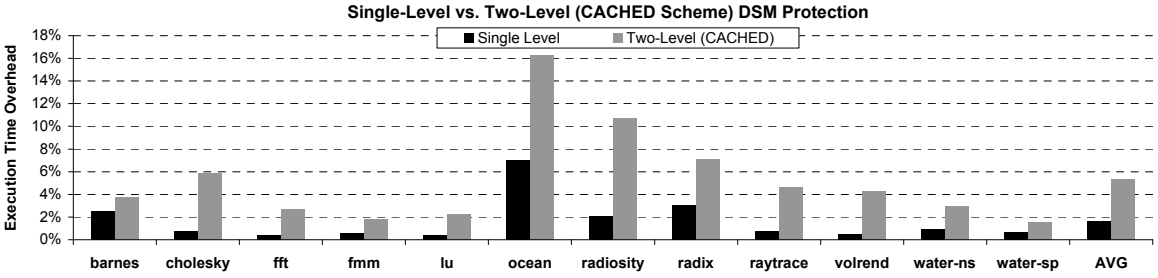


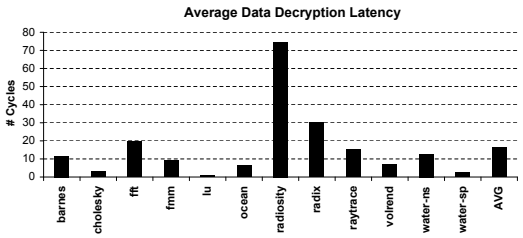
Figure 20: The execution time overhead of our single-level DSM data protection scheme versus previously proposed two-level protection using the CACHED scheme with 4-entry communication counter table [41], assuming the same security protection level.

the CACHED scheme, for example *ocean* and *radiosity* at 16.3% and 10.7% respectively. Our scheme reduces these overheads to 7.0% for *ocean* and 2.1% for *radiosity*. Equally important is the number of cases in which the execution time overheads are practically negligible. With CACHED, there are nine benchmarks that are slowed down by more than 2%, while for our proposed single-level scheme there are only two benchmarks slowed down by more than 2%. Since DSM systems are typically very pricey and they are often used to run critical applications, it is likely that performance overheads such as those seen for the worst-case applications with a two-level scheme are not tolerable. Additionally, the performance of our single-level scheme is much more stable than that of the two-level scheme. With a standard deviation of 1.9% in execution time overhead, our single-level scheme provides much more confidence to users that their applications will perform acceptably well than the previous two-level scheme with a standard deviation of 4.3%.

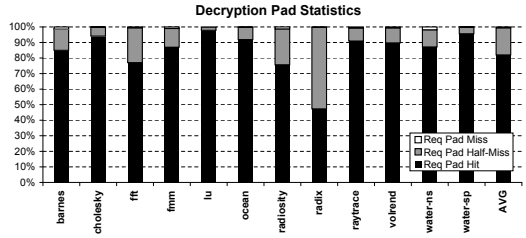
The central reason for the high performance overheads of the two-level scheme shown in Figure 20 is that cryptographic latencies may be exposed at multiple points in the critical path of a data fetch from a remote processor. More specifically, there are three points in this critical path where cryptographic delays may occur as shown in Figure 15 ((1) when a memory block requested by a remote processor is fetched on-chip by the block's home processor and decrypted, (2) when the block is encrypted again to be sent to the requesting processor, and (3) when the requesting processor receives and decrypts the block on-chip). Our data confirms this observation: on average, cryptographic latencies are at least partially exposed at point (1) 9% of the time, at point (2) 29% of the time, and at point (3) 46% of

the time. This means that roughly only $(1-0.09)\times(1-0.29)\times(1-0.46) = 35\%$ of the time full cryptographic latencies are hidden in the CACHED scheme (versus 82% of the time for our scheme – we will discuss the result in Figure 21(b) later). This shows that two-level schemes are inherently inefficient because there are too many points on the critical path where delays can be introduced.

Now that we have examined the performance of our single-level DSM data protection scheme, we will take a closer look at the reasons for its low performance overhead. Figures 21(a) and 21(b) are closely related to each other. Figure 21(a) shows the average number of cycles it takes to decrypt a fetched data block once it arrives on-chip. This latency is essentially added to the critical path of data reads. Figure 21(b) shows the percentage of off-chip data requests for which the decryption pads are fully generated (pad hit), partially generated (pad half-miss), or not generated (pad miss) when the requested data arrives on-chip. If a pad is fully generated, the decryption latency is totally hidden, while if it is partially generated then the latency is partially hidden.



(a) Decryption cycles (not hidden).



(b) Pad hit/miss percentage.

Figure 21: Performance overhead source.

For most applications, the average decryption latency is very low, around 15 cycles or less, compared to the full 80 cycles of the AES engine. Only two applications have large average decryption latencies, *radiosity* and *radix*. For *radix* this is explained by Figure 21(b) which shows that *radix* has a large amount of decryption pad half-misses compared to the other applications. For *radiosity*, one explanation of its high decryption latency is that it suffers from bursty memory request patterns on one particular processor. In our single-level scheme, the requestor is responsible for all the decryption and authentication work for its data requests. The result is that later accesses in a series of bursty ones are delayed while

earlier accesses are decrypted and authenticated. Finally, Figure 21(b) also shows that on average, 82.4% of the time pad generation latency is fully hidden, 17.1% of the time it is partially hidden, while only 0.5% of the time the latency is fully exposed.

To further explain the performance of our scheme, we present Figures 22 and 23. In Figure 22, we show the percentage of off-chip data requests for which we correctly predict the counter value for the data block. In Figure 23 we show how frequently the home node of a data block finds the requested block's counter cached in its local counter cache. This percentage is the total height of the two bars, and it corresponds to the percentage of data requests in which the home node can reply with the data's counter early to hide the decryption delay at the requestor. However, with our counter prediction scheme, if the counter has been predicted correctly by the requestor, the home node does not need to reply with a separate counter message. The gray portion of the bars shows how frequently this event occurs.

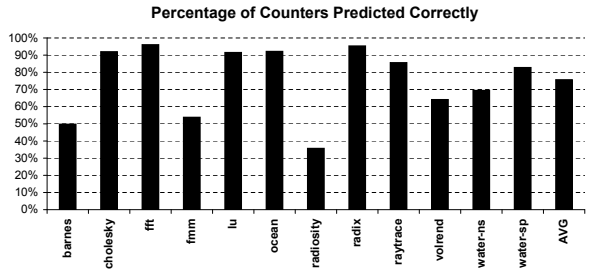


Figure 22: Percentage of requests for which the counter value is correctly predicted.

Figure 22 shows that despite its simplicity, our counter prediction scheme is very successful at correctly predicting counter values. The correct prediction rate is 76% on average, and over 90% for 5 applications. This high prediction rate benefits our scheme in two ways.

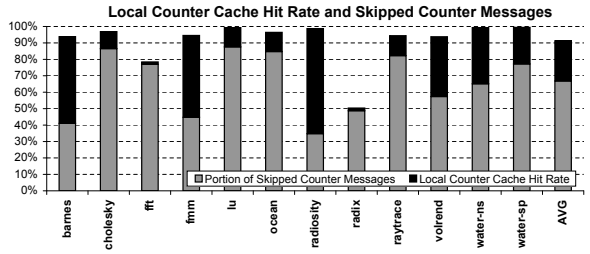


Figure 23: Local counter cache hit rate and portion of counter messages skipped due to correct prediction.

First, if counters are predicted correctly, then it is *very* likely that the decryption pad is fully pre-generated before it is needed, thus fully hiding the decryption latency. Also, as shown in Figure 23, we can eliminate a large number of separate counter messages from the home processor to the requestor. This reduces the pressure placed on interconnect bandwidth, because a separate counter message requires more overhead than simply including the counter value with the data of the reply message. Figure 23 also shows that, when counter prediction fails, most of the time we can still hide the decryption latency by forwarding the correct counter. This figure shows that a block's counter value can either be predicted or sent early over 90% of the time in most cases, and 91% of the time on average.

The final comparison we make between our single-level scheme with the previously proposed two-level scheme is on the AES unit bandwidth utilization shown in Figure 24. This figure shows that due to the reduced amount of cryptographic work, for most applications we observe a large decrease in AES utilization with our single-level scheme. For all but two of the applications, we use the AES unit 30% less than in the two-level scheme, and for some applications this savings is closer to 40%. This result shows that we provide secure data encryption and authentication in a DSM system with fewer cryptographic operations required compared to a two-level scheme.

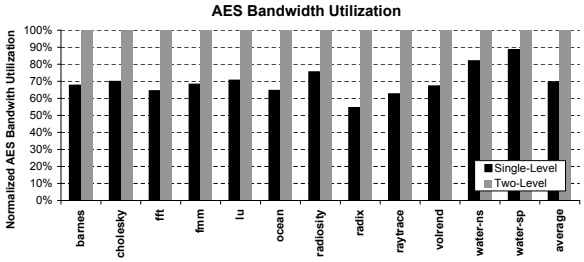


Figure 24: The normalized AES bandwidth usage of single-level vs. two-level DSM data protection.

4.6.2 Sensitivity Analysis

In Figure 25, we show how our single-level DSM data protection scheme performs as the number of processors in the DSM system increases. Again, the performance is shown as execution time normalized to a DSM system with equivalent configuration but with no support for data protection.

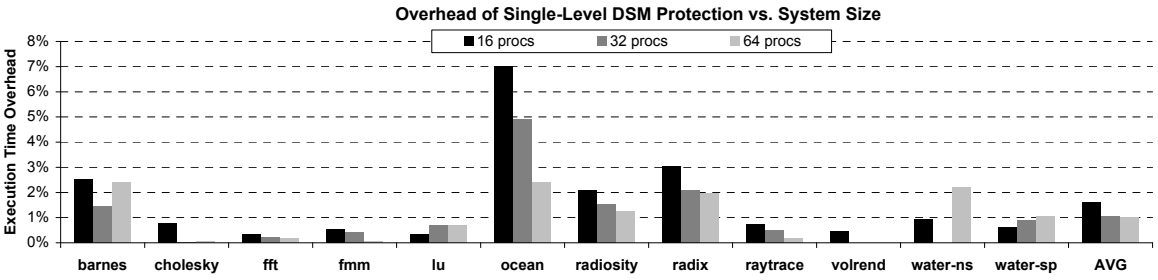


Figure 25: Execution time overhead of our single-level DSM data protection scheme across system size.

It is clear from this figure that the overheads of our protection scheme remain low as the number of processors increases. In fact, in most cases and on average the overhead decreases with respect to the system size. On average, the overhead goes from 1.6% on 16-processor DSM to 1.0% for both 32-processor and 64-processor DSM system. The worst-case overhead also improves significantly from 7.0% on 16 processors down to 4.9% on 32 processors and 2.4% on 32 processors. With a larger DSM size, there are more remote data requests because data is scattered around more nodes. However, at the same time communication latencies increase with the number of processors since data must travel more hops across the interconnection network on average, making the impact of cryptographic latencies less significant relative to the total remote data fetch latencies. The figure shows that in general, the increase in inter-node communication is the more important factor, resulting in reduction in execution time overheads to just 1% as the DSM size increases.

Figure 26 shows the overheads of our single-level DSM data protection scheme as the size of the L2 cache varies from 128KB to 256KB (our baseline size) to 512KB. The performance is shown as execution time normalized to a DSM system with equivalent configuration but with no support for data protection.

As shown in this figure, generally the overheads are reduced as the cache size increases, for example the average overhead is 2.5% with 128KB L2, 1.6% with 256KB L2, and only 0.8% with 512KB L2. The reason is that larger caches in general reduce the amount of traffic to memory, and thus the amount of data encryptions, decryptions, and authentications that our single-level DSM protection scheme must perform. Note that even with a 128KB L2 cache size, where our scheme is stressed more heavily, the overheads remain low, with an

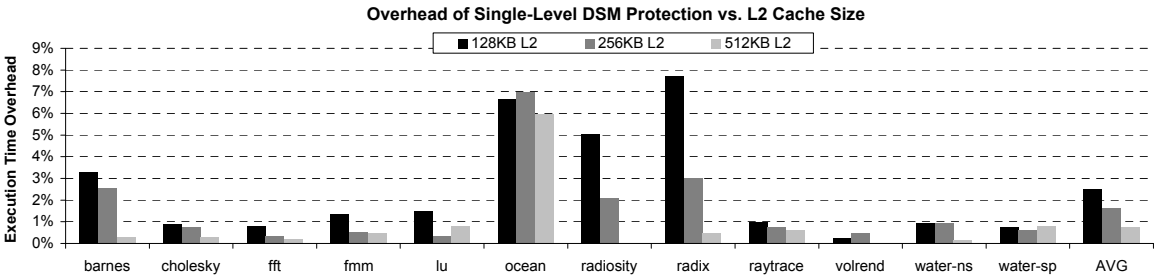


Figure 26: Execution time overhead of our single-level DSM data protection scheme across L2 cache size.

average of only 2.5% and a maximum of only 7.7%. This indicates that our scheme will perform well even when it is heavily stressed in scenarios where the amount of off-chip data communication is large.

4.7 Discussion

In this chapter, we have proposed a single-level data encryption and authentication scheme to protect the confidentiality and integrity of data in Distributed Shared Memory multiprocessors that use a point-to-point interconnect. Our scheme reduces the amount of cryptographic work by a factor of three compared to a previously proposed two-level approach, and reduces the average execution time overhead by a factor of 3.3× (from 5.3% to 1.6%). In addition, our single-level scheme reduces the performance penalty of applications that suffer from interolable overheads with a two-level scheme down to a much more acceptable level. The overheads due to our single-level scheme are also much more stable than those seen with a two-level scheme. Our approach requires only relatively minor modifications to secure processors used for uniprocessor systems, can scale to any number of processors, and can work on a wider variety DSM systems than prior approaches. We found the overheads tend to decrease when the DSM has more processors, and when each processor in the DSM has larger caches.

Security support for Distributed Shared Memory multiprocessors is an important factor which sweeps away many concerns regarding the security of utility computing. As the industry matures further, software development, distribution and execution pipeline has been divided into independent tasks and handled by different companies which specialized in their

core areas. This operation model helps improve companys expertise in their core businesses and drive innovation in the software industry. For example, Microsoft introduced its Azure Service Platform recently which enables third-party developers write applications that run partially and/or entirely in a remote datacenter with a platform and set of tools. Third-party developers will be liberated from the tedious work for building the infrastructure and planning for computing resources. Instead, they can focus their work in the application layer with the infrastructure support provided by the hosting company.

For software companies, the most valuable asset is the piece of software bits they own. It is reasonable that Intellectual Property (IP) protection questions have been raised again and again by potential users of utility computing. For many of them, it is an uneasy decision that they would run their software application in a remote datacenter of which they do not have much control. The situation would be even trickier if they would run their software application in a same infrastructure with their competitors or even the host company can be a potential competitor of their businesses. With the security support for Distributed Shared Memory systems, such concerns can be swept away easily. Secure DSM systems bring in another layer of protection which is provided by processor manufacturers.

Our proposed scheme for secure DSM systems reduces performance overhead both on average and for worst case application. Moreover, it helps shift the cryptographic operations to the actual consumer of the encrypted data. The single-level data encryption and authentication scheme avoids the hot-spot situation when multiple consumer processors request the data blocks from the same host processor. The design of single-level data encryption and authentication scheme not only trumps the two-level design in performance, but also scales much better when the size of DSM systems increases.

In addition to the performance and scalability benefits of the single-level design, it also uses less power for cryptographic operations in comparison with two-level schemes. It is pretty straight forward to see why single-level scheme spends less power than two-level schemes. For the same operation, single-level data encryption and authentication scheme only introduces one cryptographic operation while two-level schemes demand three of the same operations. There are also intrinsic savings in storage due to the omission of two

cryptographic operations. Power usage may be a trivial issue for personal users but it is a far more important concern for host companies of utility computing. Companies which maintain large clusters of computers usually build their datacenters in areas where energy cost is low. The savings in power usage in single-level data encryption and authentication scheme can be a decisive factor for utility computing host companies to control their operating cost.

Overall, the single-level data encryption and authentication scheme proposed here makes utility computing more trust-worthy and easier to be adopted by the industry.

CHAPTER V

REDUCING OFF-CHIP COMMUNICATIONS THROUGH CACHE OPTIMIZATION

5.1 Motivation

In previous chapters, we proposed efficient schemes for memory encryption and authentication for both the uni-processor system and the multi-processor system. The focus of the schemes is about how to protect the off-chip communications effectively. To move the cryptographic operations off the critical path, we proposed techniques to parallelize the cryptographic computation latency with the memory access latency.

However, we also notice that we can reduce the performance overhead in secure architecture by minimizing the off-chip communications. The cache is used to store data or code temporarily on-chip to avoid memory accesses when the processor needs to access the same block in the near future. With the growing gap between the processor cycle time and memory access latency in recent years, the on-chip cache becomes more and more important. In secure architecture, the on-chip cache becomes even more important due to the additional cryptographic operations for off-chip data accesses. In this chapter, we aim to reduce off-chip communications through on-chip cache optimization. As a result, we can improve performance of secure systems through reducing cryptographic operations.

We observe that not all recently accessed blocks tend to be reused while in the cache. We also notice that the probability of this reuse is a predictable property of the instruction that triggers the cache miss which brings the block into the cache. Armed with these observations, we propose a novel reuse-based cache allocation policy that omits cache allocation for blocks that are predicted as unlikely to be reused.

Two key mechanisms are needed to implement this policy. First, we need a reuse detection mechanism to train a reuse predictor, so we add a single "reuse" bit to each cache line. This bit starts off as zero when the block is inserted in the cache, and is set to one when the block is accessed again. When the block is replaced from the cache, its reuse bit

is used to train the predictor.

The second mechanism is the reuse predictor itself. This predictor is trained each time a block is replaced, and is consulted on a cache miss. A very important observation for our scheme is that the misprediction penalties are biased, and favor predicting that a block *will* be reused. This is because an incorrect "no-reuse" prediction leads to not placing the block in the cache and a future cache miss when it is re-accessed. A wrong "reuse" prediction leads only to replacing another cache block from the cache. In this case, the replacement policy will replace a block that it thinks is unlikely to be reused soon, so a wrong "reuse" prediction only occasionally leads to cache misses.

As a result of this observation, we want our reuse predictor to be heavily biased towards giving a "reuse" prediction—it should predict "no-reuse" only when reuse is very unlikely. Ideally, the predictor should indicate *how unlikely* reuse is, so we can take more or less risk depending on the prediction. Unfortunately, existing predictors and confidence estimators based on saturating counters are not suitable for such prediction because they tend to provide maximum-confidence predictions even when one outcome is only marginally more likely than the other, as we will discuss in detail in Section 5.3. Even biased probabilistic counters recently proposed by Riley and Zilles [40] tend to saturate one way or the other depending on whether the probability of event (non-reuse, in our case) is above or below a pre-set threshold. With this in mind, we develop a novel predictor we call *probability estimator* to predict the actual probability of a particular outcome, i.e. a predictor whose state can directly be used as an estimate of the probability itself.

Overall, this work makes several key contributions. First, we demonstrate that reuse of cache blocks is a predictable property that can be used to guide allocation decisions and improve overall cache performance. Second, we propose a novel probability estimator that can be used to estimate the actual likelihood of an outcome, and demonstrate its effectiveness in reuse prediction. Finally, we propose a novel cache allocation policy based on reuse prediction, and show that it improves performance for both the secure systems and the systems without security architectural support. For secure systems, our scheme helps eliminate more than half of the performance overhead caused by memory encryption and

authentication and makes the architectural support for security more affordable.

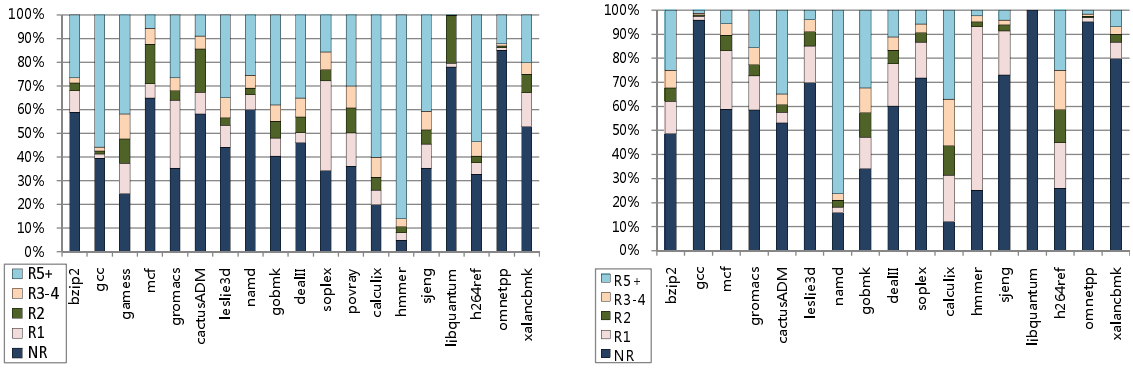
The rest of this chapter is organized as follows: Section 5.2 presents an overview of our approach. We discuss our design in Section 5.3. Section 5.4 discusses the implementation issues and the hardware cost of our scheme. Section 5.5 presents our evaluation results and insights. Finally, we conclude with Section 5.6.

5.2 Overview

On a cache miss, the block is fetched from memory and inserted into the cache to obtain cache hits on near-future accesses to this block. Our goal in this work is to omit this insertion when we predict no benefit from it - the block will be replaced from the cache before it is accessed again. We refer to our scheme as Probabilistic Approach of Selective Cache Allocation (PASCA). Prediction of reuse is related to dead-block prediction [21] and last-write prediction [22, 23], both of which predict whether or not the block will be accessed in a particular way if it remains in the cache. Dead block and last-write prediction have been proposed to improve prefetching by replacing the dead block with a prefetched one, to reduce the number of coherence interventions by self-invalidating a block that will not be reused if it remains in the cache, and to reduce vulnerability to faults via write-backs of data that will not be written again while in cache. The key common elements in these prior schemes are that 1) the block is already brought into the cache, and 2) that prediction is needed for every write in last-write prediction and for every access in others. In contrast, our PASCA scheme prevents non-reused blocks from being inserted into the cache, and it only generates a prediction or needs an update on a cache miss.

Several prior schemes [17, 18, 51] also use selective cache allocation to improve cache perofrmance. Tyson et al. [51] use the profiled or predicted miss rate or an instruction to decide whether or not to allocate a cache line for blocks fetched by that instruction. Johnson et al. [17, 18] base the allocation decision on whether the access frequency of the incoming block is higher than the access frequency of the block it would replace. The key common element of these schemes are that 1) they need to track access frequencies for cache blocks, which requires activity on every cache access (hit or miss) and 2) they base

allocation decisions on access frequencies and miss rates, which are only surrogate measures for the actual likelyhood or reuse. In contrast, our PASCA hardware is only accessed on cache misses, and bases its allocation decisions directly on probability of reuse.



(a) Reuse count in 16KB L1

(b) Reuse count in 1MB L2

Figure 27: Percentage of data blocks with difference reuse frequencies in data cache

Figure 27 shows the percentage of data blocks with different reuse counts in the data caches. To generate this Figure, we collect the reuse count for each block while it is in the cache, and update our statistics when the block is replaced. Therefore, the figure directly shows percentages of cache allocations that have resulted in a particular number of cache hits. As we can observe from this figure, many cache allocations result in no reuse. Our PASCA scheme is intended to predict non-reuse and skip cache more allocation for such blocks. Two applications (*games* and *povray*) were not presented in the reuse count for L2 data blocks, because these two applications fit entirely in the 1MB L2 cache and no L2 blocks are ever replaced. Different applications exhibit different reuse patterns for both the L1 cache and the L2 cache. In some applications, such as *calculix* and *hmmer*, most blocks inserted into the L1 or L2 cache are subsequently reused. In others applications, such as *omnetpp*, block inserted into L1 or L2 are rarely reused. In several applications, reuse patterns in the L1 cache and in the L2 cache differ significantly. However, an overall observation can be made that a significant portion of data blocks inserted into a cache see no-reuse before being replaced from that cache, so it might be possible to improve cache performance by skipping cache allocation (not inserting the block into the cache) for such blocks.

A disadvantage of skipping cache allocation for NR-predicted blocks is that the penalty for each misprediction is an extra cache miss. As a result, we require a *very high* probability of NR in order to skip an allocation. Unfortunatelly, ordinary N-bit saturating counters that are commonly used for predictors are not suitable for this kind of prediction because they tend to give strong predictions (counter is saturated) even if one outcome is only slightly more likely than the other. The first modification we make to these counters is to use biased probabilistic counting similar to that used in Riley and Zilles [40]. We decrement the predictor's counter each time the replaced line is found to have been reused, but a line that has not been reused results in a counter increment only with some lower-than-one probability P. For example, is P is 1%, the counter will only tend to go up (predict non-reuse) if non-reuse is about a hundred times more likely than reuse. By choosing P, we can change the amount of risk we are willing to take. Obviously, $P = 0$ results in no risk - we always predict reuse and our scheme behaves like a normal cache. Conversely, $P = 1$ results in ordinary saturating counting and our scheme takes too much risk. Values of P between 0 and 1 provide different levels of risk, but it is still difficult to find the right threshold probability P such that we can always allocate when probability of non-reuse is lower than P and always skip allocation when the probability is above P. Ideally, we should have a predictor that predicts the probability itself, not whether or not the probability is below or above a threshold. This would allow us to control the risk by *probabilistically* skipping allocations - the higher the predicted probability of non-reuse of a block is, the more likely the scheme should be to skip that allocation for that block.

Another problem in implementing our allocation policy using a threshold is that the predictor is only trained on cache replacement. In effect, the predictor is never trained using blocks whose allocation has been skipped. As a result, once the predictor starts consistently predicting non-reuse for a set of blocks, there is no way to change that if the behavior changes. This problem is also avoided by probabilistic skipping of allocations - even for very high probability of non-reuse, the rare allocations that still occur for those blocks allow the predictor to be trained.

To allow prediction of probability itself, we develop a predictor which we call *probability*

estimator. It modifies the probabilistic counter to decrease the probability of counting up as the value of the counter grows. With this modification, the counter's value no longer tends to saturate up or down depending on whether the probability of non-reuse is higher or lower than a given threshold probability P. Instead, the value of the counter tends to stay in the vicinity of the value where the probability of counting up corresponds to the probability of non-reuse. As explained before, the main difference between our probability estimators and probabilistic counters proposed previously by Riley and Zilles [40] is that our probability estimator allows direct prediction of the probability without having to pre-select a specific threshold. This will allow our PASCA scheme to make allocation decisions probabilistically, instead of all-or-nothing decisions that can be made with threashold-based predictors. Our probability estimator will be discussed in more detail in Section 5.3.

With the probability estimator, we can bypass cache allocation more frequently when the probability of reuse is extremely low, bypass seldom when the probability of reuse is merely low, or don't take any risk (always allocate) when the probability of reuse is significant. This also results in training the predictor more often when the non-reuse outcome is less certain.

Qureshi et al. proposed adaptive insertion policies [38] which also change the insertion policy rather than modify the victim selection process in the replacement policy. In [38], data blocks are firstly inserted into the LRU position and are only promoted to MRU position on reuse. To mitigate the problem that some applications do prefer traditional LRU algorithm over the new insertion policy, Quershi et al. make use of dynamic insertion policy which pick the better performing policy from LRU and new insertion policy in flight. In contrast, PASCA directly predicts the reuse possibility for each data block. The decision of skipping cache allocation is made independently for each data block. Moreover, PASCA is more aggressive when non-reuse of the block is predicted with high confidence. The data block will not be inserted at all in comparison with insertion into LRU position in [38].

For secure systems, we face an additional challenge when the security related blocks such as counter blocks and MAC blocks are stored together with the data blocks in the L2 cache. The cache behavior for security related blocks usually vary significantly from

that of normal data blocks. However, security related blocks can be stored at any set and thus fail schemes such as set dueling in [38]. It is imperative to predict reuse and make allocation decision for each data block independently. Our PASCA scheme is suited for this requirement.

In addition to schemes that directly tackle cache replacement [16, 36, 49] and allocation decisions, several studies have focused on improving the cache's effective associativity. Hallnor et al.'s fully associative cache [11], Qureshi et al.'s V-Way cache [37], Peir et al.'s adaptive cache [35]and Bodin and Seznec's skewed associativity [5, 42] are some representatives for this approach. Another group of studies in the area looks at improving the effective capacity of the cache, such as line distillation in Qureshi et al. [39] and adaptive cache compression in Alameldeen et al. [1]. As we will discuss later in section 5.5.2, our scheme works for different cache sizes. The major source of performance improvement in our scheme is the skipped allocation for data blocks that would not be reused if they were inserted into the cache. Moreover, our approach is orthogonal to distillation and compression schemes, and should work well together with them to avoid wasting the available effective cache capacity on caching blocks that are unlikely to be reused during their lifetime in the cache.

5.3 Design

5.3.1 Reuse Prediction Table

Like prior predictor-based policies for cache allocation, our PASCA mechanism uses the instruction's address as the index for reuse prediction. Because non-reused blocks are far more numerous than the instructions that bring them into the cache, a table indexed by the instruction address is significantly smaller than if we had a table indexed by block address.

Once the data block arrives on-chip, we first check the prediction table using the instruction's address (PC) as the index into the predictor table. The prediction table is an untagged array of probability estimator entries, so lower-order bits of the instruction address are used as the index, and the probability estimator's value is simply read out (no tag check). This predicted probability of reuse is then used to decide whether or not to insert the block into the cache. If the block is not inserted into the cache, the data is

directly supplied to the processor. In secure systems, the L2 data cache miss may also trigger the memory access of the corresponding counter value if it is not cached on-chip. In this scenario, the data block and the counter block should avoid using the same index for the prediction table since their cache behaviors usually vary significantly. To solve this problem, counter blocks use the hash value of the instructions address as the index into the prediction table and thus train a different prediction table entry for its probability of reuse.

Initially, all entries in the prediction table are initialized to the state of predicting frequent reuse. Therefore, cache allocation is not skipped initially and the cache works like the normal cache. For every cache line, we maintain a reuse bit which is set to 0 when the cache line is first allocated. The reuse bit will be set to 1 when the cache line is actually reused later. When the data block is replaced, we check the corresponding reuse bit. If the bit is 1 (block has been reused) we will train the corresponding entry in the prediction table towards reuse. If the reuse bit is still 0 when the block leaves the cache, we will train the prediction table towards no-reuse. After the program runs for a while, the prediction table will be trained enough to begin predicting non-reuse for instructions that tend to bring in non-reused blocks.

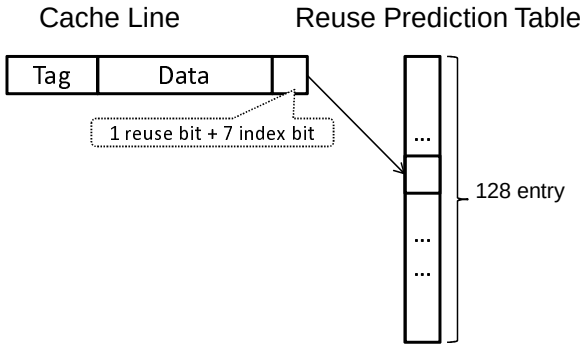


Figure 28: Reuse Prediction Table

Fig 28 illustrates the structure of the reuse prediction table. Recall that the index into the predictor table is the address of the instruction that suffers a miss and brings the block into the cache. This address is available at the time of the miss when the predictor lookup is done. However, the predictor is updated when a block is replaced, and the entry that should be trained is the entry that corresponds to the instruction that brought the

block into the cache, not the instruction that suffers a miss and causes the replacement. Therefore, the state of each cache block is extended to store the predictor table index for the instruction that brought the block into the cache, and this index is then used when training the predictor on block replacement. The additional bits are relatively few because the number of predictor entries is small (7 index bits for a 128-entry table). These additional bits are only used during a cache miss: when an instruction suffers a miss and replaces another block from a cache line, that line's index bits are read to update the correct predictor entry, and then they are written with the current instruction's index bits to ensure that a future predictor update can be correctly performed. Because these additional index bits are not needed to handle cache hits, they do not need to be part of the cache's fast and multi-ported tag array. Instead, they can be kept as a small, separate array that only needs to be fast enough to support lookups and updates during the *cache miss* latency. this array can also be single-ported because its access bandwidth is one read and one write per cache miss.

5.3.2 Probability Estimators

In our probability-based reuse prediction scheme, each entry in the reuse prediction table contains 4 bits. The first 3 bits are used as the probability estimator and the last bit is used for hysteresis. The purpose of the hysteresis bit is the same as the purpose of the lower-order bit in a 2-bit saturating counter predictor: it serves to prevent the probability estimator from changing its prediction too easility. Figure 29 illustrates the prediction table entry and how its value is updated and interpreted as a probability estimator.

The lowest state (0) indicates that the data block is likely to be reused within its lifetime in the cache. The highest state (7) indicates that the data block is very unlikely to be reused. The probability estimator is trained differently in different directions because we want it to "learn" quickly in the direction of making allocations and take its time before it starts deciding not to make allocations. To accomplish this, the predictor's value is decremented by 1 each time a reused (R) data block leaves the cache. In contrast, when a non-reused (NR) block is replaced the prediction entry value is incremented only with some probability,

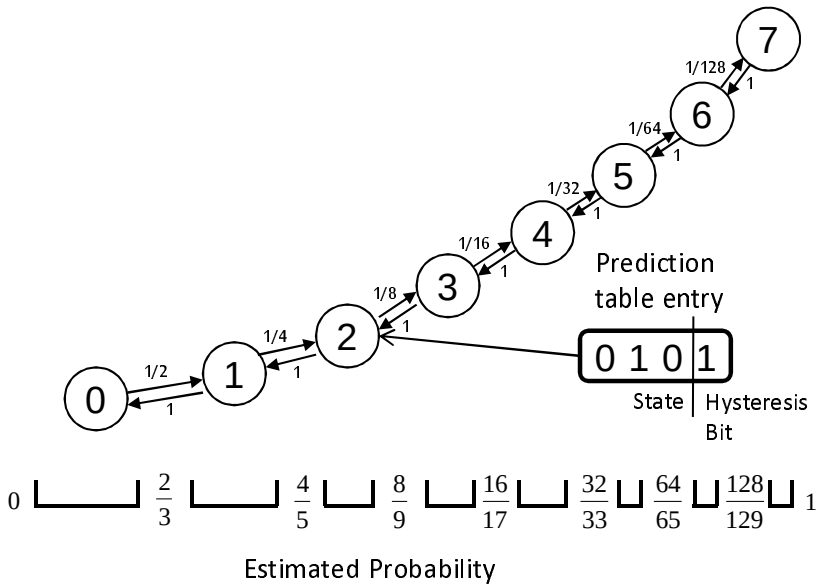


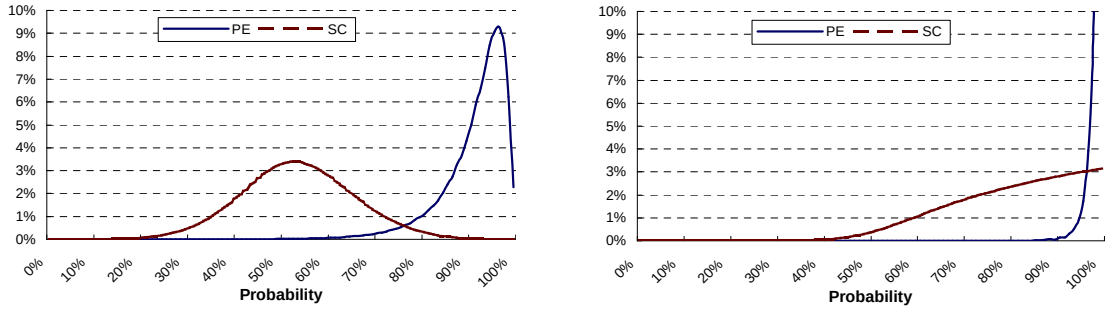
Figure 29: Using a prediction table entry as a probability estimator.

and this probability depends on the current state of the estimator. As shown in Figure 29, if the prediction value is 0000 or 0001, it is incremented with 50% probability on a non-reused block replacement. If the prediction value is 0010 or 0011, it is incremented with 25% probability on a non-reused block replacement, etc. The higher the current state of the estimator, the less likely it is to transition to a higher state.

To implement this probabilistic approach, we use a hardware random number generator, either by implementing an actual random number generator in hardware or by amplifying the thermal noise already present in the processor. For state K ($0 \leq K \leq 7$), a (K+1) bit random number is generated on a non-reused block replacement. The entry value is incremented only if all (K+1) bits are zeroes. In general, the probability for moving from state K-1 to state K is $\frac{1}{2^K}$. We note that state 7 is the highest state and the estimator never goes up from that state, so the uniform-distribution hardware random number generator we use only needs 8 bits.

This approach to probability estimation allows us to distinguish between different high levels of confidence for a NR prediction - when the probability of NR is less than 66%, the estimator will tend to be in state 0, and it will tend to be in state 7 only when the probability of NR is more than 99%. As a result, the value of the probability estimator

indicates how much risk we are taking if we skip allocation.



(a) Counter state 4.

(b) Counter state 7.

Figure 30: Comparison of probability distribution.

Figure 30 shows the distribution of the actual observed probability of NR for several different values of the probability estimator. We also show these distributions for the normal 3-bit saturating counter for comparison. With saturating counters (SC in the figure), the most confident NR prediction (state 7) can obviously be taken as an indication that the actual probability of NR is above 50%. However, many of these state-7 predictions are for NR probabilities of only 60%, 70% or 80%. Because this predictor does not distinguish well between different levels of high NR probability, it is not very helpful when deciding how much risk to take. On the other hand, the probability estimator's (PE) state 7 clearly indicates a very high NR probability - most of the time this prediction is made, the actual NR probability is likely above 99% (1 in 100 chance of making a misprediction) and we can take a lot of risk. When the probability estimator is in state 4, the indicated actual probability is likely above 90% (1 in 10 chance of making a misprediction), so we need to take some risk, but less than for state 7.

5.3.3 Probabilistic Cache Line Allocation

Our probability estimators tell us approximately how likely a data block is to be useless if it is inserted into the cache. If the indicated probability is too small (reuse is too likely), we will allocate a cache line for the block to avoid the risk of a cache miss when reuse does happen. For non-reuse probabilities that are high enough, we want to take some risk and skip allocation of some such blocks, but insert enough of them into the cache to avoid some risk and to keep training our predictor. For higher non-reuse probabilities, we are willing

to take more risk and skip cache allocation for more such blocks.

To achieve this inter-dependence between risk-taking and non-reuse probability, we use the random number generator [1] to make allocation decisions probabilistically, where the probability of allocation depends on the value of the probability estimator. When the estimator is in state 0, we take no risk (reuse is too likely) and never skip allocation. When the estimator is in state 1, we skip allocation 50% of the time, etc. In general, when the estimator is in state K, we insert the block into the cache with probability of $\frac{1}{2^K}$. Note that there is a small chance (slightly less than 1%) of cache allocation even when the probability estimator is in state 7 (NR is extremely likely). These infrequent cache allocations result in a minor loss of opportunity, but are needed to train our probability estimators in case the program behavior changes. For example, if there is a dramatic change and reuse becomes very likely while the estimator is in state 7, an allocation will still occur within the next few hundred cache misses, reuse will be detected, and the estimator moved to lower state. This causes the allocation to become more likely, which causes reuse to be detected soon, the estimator moves rapidly to even lower states and, eventually, to state 0 where allocation is no longer skipped. On the other hand, if the predictor is in state 0 and reuse becomes very unlikely, a few non-reuses will be detected and one of them will increment the prediction entry value from 0000 to 0001 (still state 0), then from 0001 to 0010, etc. To reach value 1110 from 0000, the predictor needs to see a considerable number of non-reuses (nearly 512) in a row without seeing any reuse. Once it has a value of 1110, it takes an average of another 256 non-reuses to move the value up to 1111. If the predictor sees a reuse, its value will be decremented immediately. Thus, the predictor tends to be in states 1110 and 1111 only if non-reuse is at least 128 times as likely as reuse. In contrast, the predictor tends to have a value of 0000 and 0001 if the probability of non-reuse is 66% or less: it takes a single reuse to undo the increment which occur on average for every second non-reuse. Overall, our predictions rapidly change towards normal allocation, but move slowly towards skipping more allocations, and the state of the probability estimator can directly indicate

[1] This is the same uniform-distribution hardware random number generator (RNG) that is used to update the counter value in the probability estimator. A similar RNG is needed for probabilistic counting used by Riley Zilles [40].

a wide range of non-reuse probabilities. This behavior is consistent with our intentions because doing allocation is much less risky than skipping it, so we need to rapidly prevent high-risk behavior but take on extreme risk only after we achieve extreme confidence that the risk is warranted.

5.4 Implementation

5.4.1 Prediction Tables

In our design, each data cache (L1 and L2) has its own prediction table to predict cache reuse within that cache. Although the accesses to L1 and L2 caches are caused by the same stream of accesses from the processor, different cache sizes result in different reuse patterns that should be tracked separately. For exmaple, a block may never be reused in the smaller L1 cache because it is always replaced before it is re-accessed. In the larger L2 cache, however, this block can stay in the cache longer and see frequent reuse because L1 misses regularly occur. Conversely, a block that tends to be very frequently used in L1 may not see reuse in L2 because accesses do not reach the L2 cache.

We note that our reuse prediction tables are only accessed on cache misses, when they are needed to make an allocation decision for the incoming block and, in case of allocation, to train the entry indicated by the replaced block. As a result, each prediction table only needs one port even when the corresponding cache is multi-ported. These tables are also not performance critical - the accesses to these tables occur in the shadow of a cache miss, so an entire cache miss latency is available to make a prediction and, if the decision is to allocate, train the entry of the block that will be replaced. This makes it possible to use cheaper and slower logic and state in implementing these tables. Finally, these tables are directly indexed, with relatively few (128) small (4-bit) entries, so they should not occupy significant chip area.

5.4.2 Skipping Cache Line Allocation

When a block is fetched into the L2 cache and our scheme decides to skip allocation, we forward to the L1 cache the entire block or a part of it, depending on whether the L1 block size is the same or smaller then in L2. Similarly, if the L1 cache allocation is skipped,

additional logic is needed to forward the block's data to the processor without inserting the block into the cache.

5.4.3 Collaboration of L1 and L2 Cache

The prediction tables for L1 and L2 cache work independently. However, the allocation decisions can be made collaboratively, so that L1 and L2 cache can use each other as a safety net. If the data skips allocation in the L2 cache, we increase the probability of its allocation in L1. The idea is to reduce the probability of skipping allocation of the block in both caches to avoid having a miss with a full main memory access latency in case of a misprediction. Therefore, the collaborative scheme amends the cache allocation probabilities in the following way: if the data block has skipped the cache allocation in the cache levelN+1 (e.g. the L2 cache), the probability of allocation in cache level N (e.g. the L1 cache) is increased 4 times. To implement this behavior, if allocation has been skipped in the L2 cache, in the L1 cache we bias the allocation decision by using a value from the L1 probability estimator minus 2. This still skips most allocations if the non-reuse probability estimate in L1 is very high, but helps avoid case when estimators in both caches have weak non-zero values but a "lucky" combination of random generator values in both caches reuslts in both caches selecting to skip allocation.

5.4.4 Suppressing Continuous Allocation Skips

If the cache size is very small compared with the application's footprint, the local miss rate for this application is very high. In a situation like this, the L2 cache keeps replacing old cache lines with new cache lines and most data blocks are unlikely to be reused within their lifetimes in the L2 cache. In our L2 reuse prediction table, most entries are thus trained to show a high confidence that the data block is unlikely to be reused. As a result, PASCA begins skipping allocation for most blocks. However, recall that our reuse probability estimator is trained only by blocks that were allocated in the cache. As a result of skipping most block allocation, the training of the probability estimator becomes rare and it is sluggish to adjust itself when the cache behavior changes. This can be a problem because skipped cache allocations themselves may result in a change in caching behavior:

data sets that didn't fit in the cache may fit in the cache once the other non-fitting data stops coming into the cache. To help understand this behavior, consider a cache with 4 blocks that uses the LRU replacement policy and sees the sequence of accesses shown in the first row of Table 5.

Table 5: Reuse behavior can change as a result of skipped allocatins

Access	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
If we always allocate a block on a miss:															
Hit/Miss	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
If we skip allocation of block A and allocate others:															
Hit/Miss	M	M	M	M	M	M	H	H	H	H	M	H	H	H	H

Reuse distance of this patern is 5, so all accesses are misses (second row of the table). Thus, PASCA's reuse probability estimator will be trained to predict a very high probability of non-reuse and, as a result, after a brief training period PASCA would start to skip nearly allocation for these blocks. Unfortunately, this does not lead to performance improvement: when allocation of a block is skipped, the next accesses to that block will still be a cache miss. Eventually, PASCA will (through random chance) elect to allocate one of these blocks (e.g. block A) in the cache anyway. Since other blocks are not allocated in the cache, this block will not be replaced and will be reused. This will train PASCA's reuse predictor to predict more reuse, and eventually it starts to allocate more of these blocks in the cache. However, this change will be slow because few allocations result in slow training of the probability estimator.

Note that, when allocation is skipped for one of these blocks (e.g. block A), the others fit in the cache and experience frequent reuse (third row of Table 5). To allow PASCA to quickly react to this and other situations when reuse patterns change from non-reuse to reuse, we must prevent long sequences of high-confidence allocation skips. For this purpose, each cache has a counter to count the number of consecutive skips that have been made. this counter is reset each time an allocation is made, and its value is subtracted from the value of the probability estimator when making an allocation decision. With this change, no more than 7 consecutive allocation skips can occur even if all probability estimators are at the highest value (7). In other words, at least every eight cache miss will result in

allocation and offer an opportunity to train probability estimators.

5.5 Evaluation

We use SESC [15] to model a modern desktop machine with the configuration as shown in table 6. The L1 data cache and the L2 cache each have a 128-entry table with a 4-bit probability estimator in each entry. Each cache line in the L1 data cache and L2 cache has been extended with 1 extra byte to keep the reuse bit and the 7-bit index to the prediction table. Numerous other parameters (branch predictor, functional units, etc.) are set to reflect an advanced modern desktop machine, and all occupancies and latencies are simulated in detail.

Table 6: Processor Configuration

Parameter	Value
CPU	two-issue out-of-order
ROB size	100 entry
L1 Cache	split I&D caches; each with 16KB 2-cycle 4-way set associative
L2 Cache	unified 1MB 16-cycle 4-way set associative
Memory	200 cycle latency
Bus	128-bit wide running at 600MHz

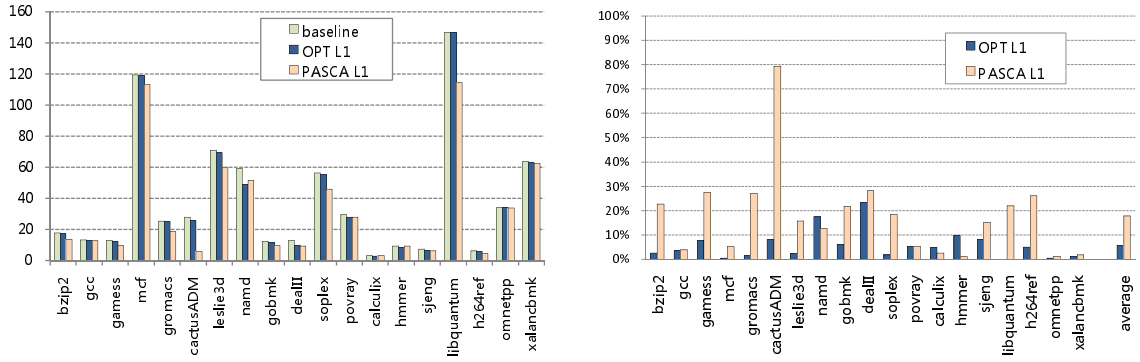
Performance results in our evaluation are shown as normalized instructions-per-cycle (IPC), where the normalization baseline is a system with normal cache allocation and replacement policies.

We use the SPEC CPU 2006 benchmarks [48] for our simulation. For each benchmark, we use its reference input set, in which we fast-forward the initial phase and then simulate 2 billion instructions in detail.

5.5.1 PASCA vs. Optimal replacement policy

Figure 31 shows the comparison of optimal replacement policy (OPT) [3] and PASCA when both schemes are used on L1 cache independently. We show both the actual numbers of L1 MPKI and the percentage of reduction in L1 MPKI for all applications.

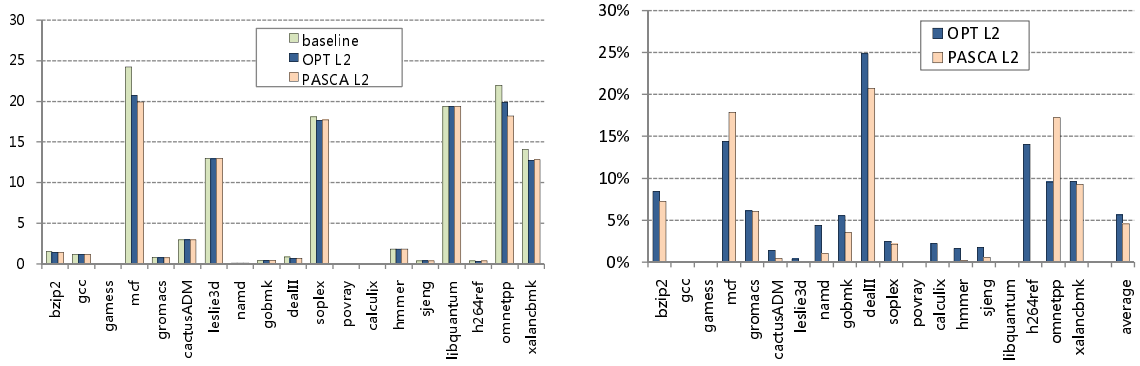
As shown in figure 31, PASCA reduces L1 MPKI more effectively than OPT. For most applications, PASCA delivers a much higher reduction in MPKI compared with OPT. Moreover, OPT does not help all the applications effectively to reduce the L1 misses. Several



(a) L1 MPKI (b) Reduction in L1 MPKI

Figure 31: Compare OPT vs. PASCA in L1 Cache

applications such as mcf, gromacs, leslie3d and soplex gain little help by using OPT but benefits significantly by using PASCA in the L1 cache. Application cactusADM exhibits the largest difference, where OPT only helps reduce L1 MPKI by 8% while PASCA manages to reduce L1 MPKI by 80% in contrast. However, there are also one application (hmmer) in which OPT significantly outperforms PASCA. As shown in figure 27(a) which we discussed earlier in section 5.2, hmmer has a very small percent of no-reuse blocks. The major source of improvement brought by PASCA is to skip allocation of these no-reuse blocks and leave the cache space to reuse blocks. In hmmer, since most blocks are to be reused, PASCA brought little benefits here. On average, OPT reduces L1 MPKI by 6% while PASCA reduces L1 MPKI by 18% in contrast.



(a) L2 MPKI (b) Reduction in L2 MPKI

Figure 32: Compare OPT vs. PASCA in L2 Cache

We also compare OPT with PASCA when both schemes are used on L2 cache independently. Both actual number of L2 MPKI and the percentage of reduction in L2 MPKI are

shown in figure 32. MPKI in the L2 cache is much lower than L1 MPKI. Several applications such as games, namd, povray and calculix almost fit entirely in the L2 cache. Neither OPT nor PASCA is able to bring any benefit to these applications when used in the L2 cache. On average, OPT reduces L2 MPKI by 6% and PASCA reduces L2 MPKI by 5%.

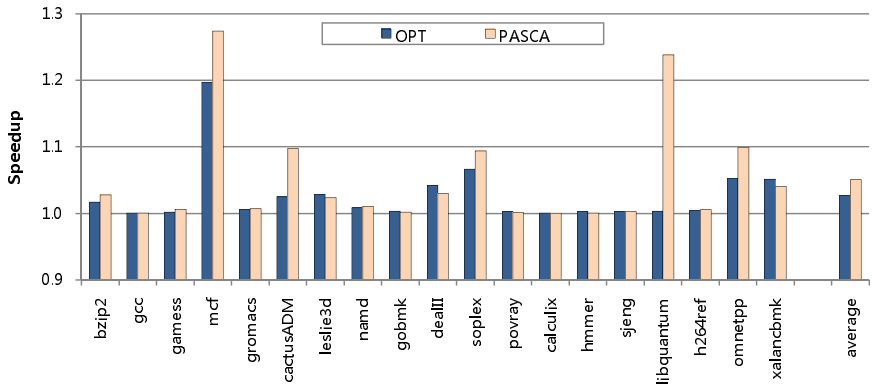


Figure 33: Performance improvement for OPT and PASCA.

Figure 33 shows the performance with OPT and PASCA when each of the scheme is used in both L1 cache and L2 cache collaboratively. All results are normalized to a baseline with LRU policy.

As shown in figure 33, our collaborative L1 & L2 scheme provides significant performance improvements in several applications, and leaves the performance of others largely unchanged. On average across all the benchmarks, we observe a speedup of 5% for PASCA used in both L1 and L2 cache. In comparison, OPT only delivers speedup of 3% when it is used in both L1 and L2 cache.

The results also show that different applications benefit differently from applying PASCA at different cache levels. For example, *mcf* benefits mostly from the L2 PASCA, *cactusADM* mostly benefits from the L1 PASCA, and *dealII* benefits from both. Another interesting application is *libquantum*. Figure 27(b) shows that libquantum hardly has any reuse blocks in L1 and L2 cache. PASCA manages to reduce a significant portion of L1 MPKI and improve performance greatly for this application.

5.5.2 Sensitivity Analysis

We also evaluate PASCA with different L2 cache size. In figure 34 we compare the performance of using different L2 cache size with and without PASCA. The baseline is a system with 1M L2 cache using normal LRU replacement policy.

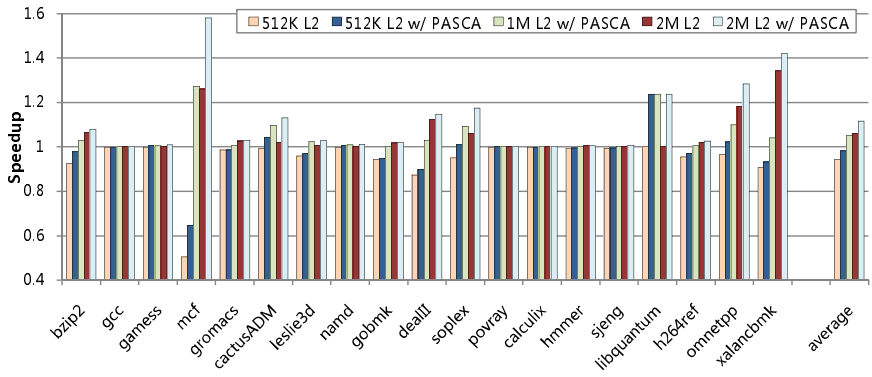


Figure 34: Performance improvement with difference L2 cache size.

There are three categories of applications as shown in figure 34. The first category is applications which benefit from neither larger L2 cache size nor PASCA. These application fit well even with a 512KB L2 cache. As a result, changes in L2 cache size or replacement policy do not affect the performance of these applications at all.

The second category of applications benefits from both larger L2 cache size and PASCA. As long as there are no-reuse data blocks which compete cache space with data blocks o be reused, PASCA will help in this case and save the cache space for blocks which are more likely to be reused.

The last category in this group of applications shows no benefit from large L2 cache size but benefits significantly from the use of PASCA. Application *libquantum* belongs to this category. As we discuss earlier in section 5.5.1, *libquantum* benefits significantly with the reduction in L1 MPKI since many L1 misses will also miss in L2 cache and go to the memory eventually. This behavior could be resulted from a sequential scan over a large memory space by the program. Most data blocks accessed will not be reused but a small working set will be reused. By using PASCA in L1 cache, we reduce L1 MPKI and improve performance consequently. There could also be a potential fourth category which only

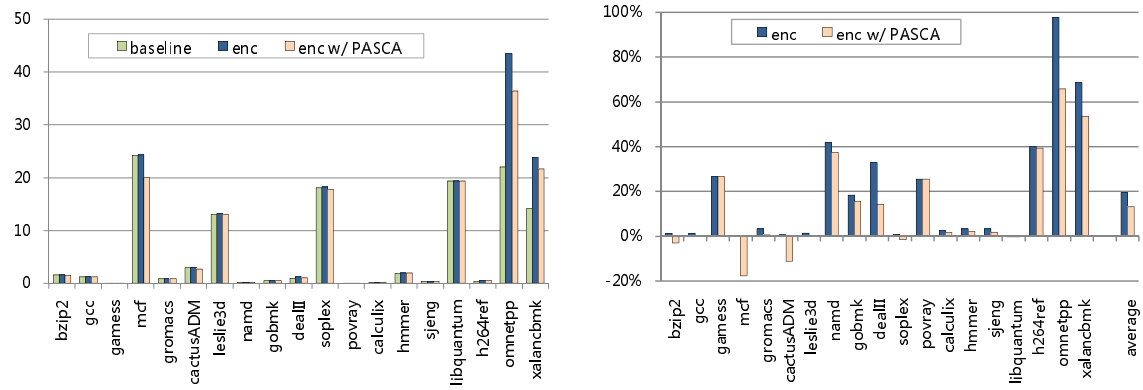
benefit from large L2 cache but not from the use of PASCA. However, we did not find any applications we simulated belonging to this category.

5.5.3 Reducing performance overhead for secure architecture

In section 5.5.2 we explore how PASCA works for different sizes of the cache. We notice that the applications can be roughly divided into two groups. One group is sensitive to changes of the cache size and this group will benefit from the use of PASCA scheme. The other group is not sensitive to changes of the cache size either because the applications in this group can fit entirely in a small cache or not memory intensive.

We use PASCA in secure architecture where the introduced memory encryption and authentication operations largely change the cache behaviors. Not only is the cache miss penalty increased, but the additional security data which needs to be stored on-chip compete cache space with normal data blocks. Previous studies for secure architecture [9, 10, 27, 50, 53] proposed several schemes to provide a copy and tamper resistant environment. Most of these schemes come with both the performance overhead and the space overhead. In this work, we implemented a secure architecture which provides support for memory encryption and authentication similar to the scheme described in previous chapters. The major difference between our implementation of secure architecture and previous work is that our scheme does not use an additional cache to store sequence numbers or authentication MACs. The additional security data (sequence number and authentication MAC) are stored together with the normal data blocks in the L2 cache and thus normal data blocks and security data blocks compete for L2 cache. Instead of using an additional on-chip cache for storing the security data blocks, we use PASCA in both L1 and L2 cache to help reduce the performance overhead caused by memory encryption and authentication.

Figure 35 shows changes in L2 MPKI when they systems use a secure architecture and how PASCA helps to reduce the increased MPKI. On average, we record a 20% increase in L2 MPKI when the system provides support for memory encryption and authentication. Such an increase in L2 MPKI is mainly resulted from the competition for L2 cache space from both the normal data blocks and the security data blocks. However, the 20% increase



(a) L2 MPKI (b) Increase in L2 MPKI

Figure 35: Using PASCA in a secure architecture.

in L2 MPKI is brought down to 12% with the help using PASCA in the L2 Cache. Moreover, using PASCA in the L1 cache also helps to retain the useful data blocks on-chip and to avoid expensive memory encryption and authentication operations.

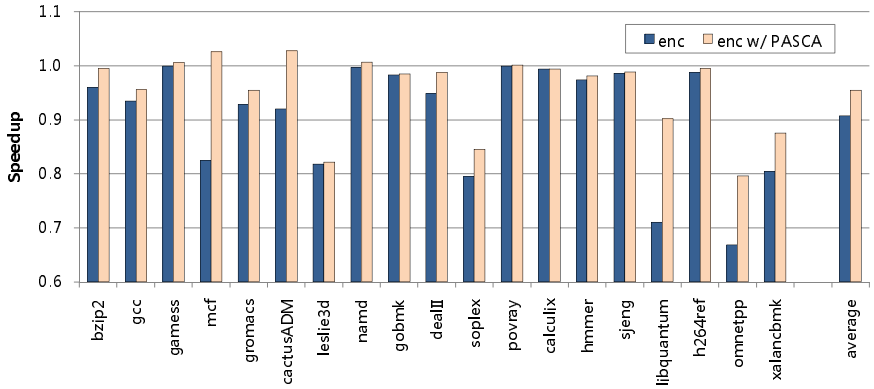


Figure 36: Reduced performance overhead for the secure architecture.

Figure 36 shows the speedup(slowdown) in SPEC 2006 benchmark applications for secure architecture and use PASCA with the secure architecture. The baseline is a scheme with neither secure architecture support nor PASCA. On average, we record 9% performance degradation for memory encryption and authentication. With the help of PASCA, we reduce the performance overhead to 4%, which cuts the overhead for memory encryption and authentication by more than half. Moreover, the worst case for using secure architecture without PASCA records a slowdown as low as 66% of the baseline scheme. With the help of PASCA, the worst case for memory encryption and authentication records a slowdown

of 80%, which is much better than the secure architecture without PASCA.

5.6 Discussion

This chapter proposes a new Probabilistic Approach for Selective Cache Allocation (PASCA). It uses a new *probability estimator* to directly predict the probability of reuse for every cache miss, and this probability estimate is then used to make an allocation decision probabilistically. This probabilistic approach allows PASCA to carefully balance the potential for performance improvement against the risk of heavy performance loss due to mispredictions, and also to allow timely training of its probability estimators.

Our scheme achieves an average speedup of 5% over SPEC CPU 2006 benchmarks, with significant speedups on some applications (up to 27% in mcf) and the worst-case of no speedup (but no slowdown, either). Our results also show that PASCA outperforms an optimal replacement policy on most applications, showing that allocation decisions can be more important than replacement decisions targeted by most prior work. The implementation cost of our PASCA scheme is very low, using in each cache a small predictor table (128 entries of 4 bits each) which is accessed only on cache misses.

Selective cache allocation is also shown to be effective in reducing performance overhead in secure architecture. Due to the extra storage overhead for counter values in counter mode based memory encryption and authentication schemes, the on-chip storage competition will be more intense. For applications which already suffer from frequent cache misses, the problem will become even worse. Selective cache allocation scheme filters out the cache allocations which are not to be reused through prediction and leave the useful data blocks and counter blocks on-chip.

Overall, PASCA is an easily implementable technique and provides consistent performance improvements (no slowdowns) in various configurations. We also believe that the key ideas presented in this study, such as our probability estimators and probabilistically managed risk-taking, can be used to guide speculation decision in other areas where a moderate potential gain for correct predictions is offset by steep penalties for mispredictions.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

This thesis presents a new combined memory encryption and authentication scheme. We first address the issues of security support in the uni-processor system environment. Our new split counters for counter-mode encryption simultaneously eliminate counter overflow problems and reduce per-block counter size, and we also dramatically improve authentication performance and security by using the Galois/Counter Mode of operation (GCM), which leverages counter-mode encryption to reduce authentication latency and overlap it with memory accesses. Our results indicate that the split-counter scheme has a negligible overhead even with a small (32KB) counter cache and using only eight counter bits per data block. The combined encryption and authentication scheme has an IPC overhead of 5% on average across SPEC CPU 2000 benchmarks, which is a significant improvement over the 20% overhead of existing encryption/authentication schemes.

We also extend our security scheme to multiprocessor computer systems which are currently widely used in commercial settings to run critical applications. These applications often operate on sensitive data such as customer records, credit card numbers, and financial data. As a result, these systems are the frequent targets of attacks because of the potentially significant gain an attacker could obtain from stealing or tampering with such data. In addition to protecting memory-processor communication where the one processor is always the one to both encrypt and decrypt data, in these systems we also have to protect processor-to-processor communication. In this thesis, we study architectural mechanisms to ensure data confidentiality and integrity in Distributed Shared Memory multiprocessors which are based on a point-to-point based interconnection network. Our approach improves upon previous work in this area, mainly in the fact that our approach reduces performance overheads by significantly reducing the amount of cryptographic operations that must be performed. Our evaluation results show that our approach can protect data confidentiality and integrity in a 16-processor DSM system with an average overhead of 1.6% and a

maximum of only 7% across all SPLASH-2 applications.

As we study the secure architecture, we observe that we can also optimize the on-chip caching mechanism to avoid performance degradation caused by memory encryption and authentication. The cryptographic operations are only applied to off-chip communications. Through optimizing the on-chip caching mechanism, we reduce off-chip communications and thus make the system more efficient and more secure. We notice that caching techniques providing better performance can benefit both counter caching and as well as the normal data caching. In this thesis, we study a new cache allocation scheme. We use a predictor to predict the likelihood of the reuse for a given data block. Data blocks which are predicted not to be reused at a high confidence level will skip allocation and leave useful cache lines stored on-chip. On average, our scheme reduces half of the performance overhead in secure architecture for CPU 2006 benchmark applications. The implementation cost of our scheme is very low, because it uses modestly sized predictors (128 entries of 4 bits each) which are only accessed on cache misses.

In conclusion, our work aims to provide efficient, secure and affordable architectural support for data security. To build a secure system, there are many other issues need to be resolved such as power efficiency, area cost, compatibility, usability and etc. However, performance overhead is usually the first obstacle to block the large adoption of the secure systems. This thesis presents several techniques to reduce the performance overhead in secure architecture and therefore make it more affordable to the users. We believe the scheme proposed in this thesis will join many other approaches in the area and help promote research and design of the emerging secure computer systems.

6.1 Future Work

Our work has shown that security support for both uni-processor environment and multi-processor environment can be affordable and efficient to implement. To bring secure architecture to life, we think there are still a few areas worth investigating to complete the picture of secure system.

First, our work and previous work assume a pre-arranged key distribution for different

components in secure systems to share the secret key. However, the implementation of such a key distribution scheme has yet been discussed in details. As the first link in building up secure systems, key distribution scheme is critical and fundamental for all cryptographic operations built on top of it. The key distribution process involves multiple parties such as processor manufacturer, software vendors and system administrators. A broken link in the trust chain can lead to catastrophic outcome of the entire secure system. The problem of key distribution is worth to be examined thoroughly to eliminate possible pitfalls as much as we can. Once we started adopting a certain key distribution mechanism, it will be very difficult to switch to another since many parties are involved in maintaining the trust relationship. The key distribution process should also be able evolve as the new requirements pop up and participants change in the process.

Second, the industry today still is not able to quantify value of secure architecture and a security benchmark is in need to compare secure processors with others. Software programmer should be involved in the creation of this security benchmark. We believe the secure architecture fundamentally changes what programmers can leverage from the infrastructure and makes development of application with high security requirement easier. A new set of security benchmark application can be developed. As a result, when consumers compare secure processors with other, they can refer to the security benchmark and realize whether the additional secure protection would make a difference in their choices of devices.

Finally, power usage and optimization for secure architecture should be studied in addition to the performance issues which are more often raised in this area. The nature of secure architecture has intrinsic impact on power usage. Cryptographic operations are additional overhead which does not exist previously. In many occasions, the power usage concerns can make a huge difference in the decision making process of the customers. For example, companies maintaining large clusters of computers migrate their datacenters to areas where energy cost is lower to reduce operating cost of their datacenters. If secure architecture demands huge power overhead, it may be intimidating for customers to adopt the secure systems. Power optimization for secure architecture is not as well studied as performance optimization in this area. We believe power issues are as important as performance concerns

to make secure systems really affordable and efficient for everyone.

REFERENCES

[1] Alameldeen, A. and Wood, D., "Adaptive cache compression for high-performance processors," in *the 31st Annual International Symposium on Computer Architecture*, June 2004.

[2] Bartholomew, D., "On Demand Computing – IT On Tap?," *http://www.industryweek.com/ReadArticle.aspx?ArticleID=10303 &SectionID=4*, June 2005.

[3] Belady, L., "A study of replacement algorithms for a virtual-storage computer.," *IBM Systems Journal*, 1966.

[4] Bellare, M., Desai, A., Jokipii, E., and Rogaway, P., "A concrete security treatment of symmetric encryption: Analysis of the des modes of operation," in *Proceedings 38th Annual Symposium on Foundations of Computer Science*, 1997.

[5] Bodin, F. and Seznec, A., "Skewed associativity enhances performance predictability," in *the 22nd International Symposium on Computer Architecture*, pp. 265–274, June 1995.

[6] Clarke, D., Suh, G. E., Gassend, B., van Dijk, M., and Devadas, S., "Checking the Integrity of Memory in a Snooping-Based Symmetric Multiprocessor (SMP) System," in *MIT CSAIL CSG-TR-470*, July 2004.

[7] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality." National Institute of Standards and Technology, NIST Special Publication 800-38C, 2004.

[8] FIPS Publication 197, "Specification for the Advanced Encryption Standard (AES)." National Institute of Standards and Technology, Federal Information Processing Standards, 2001.

[9] Gassend, B., Suh, G., Clarke, D., Dijk, M., and Devadas, S., "Caches and Hash Trees for Efficient Memory Integrity Verification," in *Proc of the 9th International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[10] Gilmont, T., Legat, J.-D., and Quisquater, J.-J., "Enhancing the Security in the Memory Management Unit," in *Proc. of the 25th EuroMicro Conference*, 1999.

[11] Hallnor, E. G. and Reinhardt, S. K., "A fully associative software-managed cache design," in *the 27th International Symposium on Computer Architecture*, pp. 107–116, June 2000.

[12] Huang, A., *Hacking the Xbox: An Introduction to Reverse Engineering.* No Starch Press, San Francisco, CA, 2003.

[13] Huang, A. B., "The Trusted PC: Skin-Deep Security," *IEEE Computer*, vol. 35, no. 10, pp. 103–105, 2002.

[14] IBM, "IBM Extends Enhanced Data Security to Consumer Electronics Products," *http://domino.research.ibm.com/comm/pr.nsf/pages/ news.20060410_security.html*, April 2006.

[15] J. Renau et al, "SESC," *http://sesc.sourceforge.net*, 2004.

[16] Jeong, J. and Dubois, M., "Cost-sensitive cache replacement algorithms," in *the 10th International Symposium on High Performance Computer Architecture*, pp. 327–337, Feb 2003.

[17] Johnson, T. L., Connors, D. A., Merten, M. C., and mei W. Hwu, W., "Run-time cache bypassing," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1338–1354, 1999.

[18] Johnson, T. L. and mei W. Hwu, W., "Run-time adaptive cache hierarchy management via reference analysis," in *the 24th Annual International Symposium on Computer Architecture*, pp. 315–326, June 1997.

[19] Kgil, T., Falk, L., and Mudge, T., "ChipLock: Support for Secure Microarchitectures," in *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Oct. 2004.

[20] Kuhn, M. G., "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP," *IEEE Transactions on Computers*, vol. Oct, no. 10, 1998.

[21] Lai, A., Fide, C., and Falsafi, B., "Dead-block prediction and dead-block correlating prefetchers," in *the 28th Annual International Symposium on Computer Architecture*, pp. 144–154, June 2001.

[22] Lai, A.-C. and Falsafi, B., "Selective, accurate, and timely self-invalidation using last-touch prediction," in *the 27th Annual International Symposium on Computer Architecture*, pp. 139–148, 2000.

[23] Lebeck, A. R. and Wood, D. A., "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *the 22nd Annual International Symposium on Computer Architecture*, pp. 48–59, June 1995.

[24] Lee, M., "Global ATM Security Alliance focuses on insider fraud," *ATMMarketplace, http://www.atmmarketplace.com/article.php? id=7154*, May 2006.

[25] Lee, M., Ahn, M., and Kim, E., "I2SEMS: Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[26] Lee, R. B., Kwan, P. C., McGregor, J. P., Dwoskin, J., and Wang, Z., "Architecture for Protecting Critical Secrets in Microporcessors," in *Proc. of the International Symposium on Computer Architecture*, 2005.

[27] Lie, D., Mitchell, J., Thekkath, C., and Horowitz, M., "Specifying and Verifying Hardware for Tamper-Resistant Software," in *IEEE Symposium on Security and Privacy*, 2003.

[28] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., MItchell, J., and Horowitz, M., "Architectural Support for Copy and Tamper Resistant Software," in *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[29] Lipmaa, H., Rogaway, P., and Wagner, D., "Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption." http://csrc.nist.gov/ CryptoToolkit/modes/proposedmodes/ctr/, 2000.

[30] Maxim/Dallas Semiconductor, "DS5002FP Secure Microprocessor Chip," *http://www.maxim-ic.com/quick_view2.cfm/qv_pk/2949*, 2007 (last modification).

[31] McGrew, D. A. and Viega, J., "The Galois/Counter Mode of Operation (GCM)." http://csrc.nist.gov/Crypto Toolkit/modes/ proposedmodes/gcm/, 2004.

[32] Merkle, R. C., "Protocols for public key cryptography," *IEEE Symposium on Security and Privacy*, pp. 122–134, 1980.

[33] Merkle, R., *Secrecy, authentication, and public key systems*. PhD thesis, Department of Electrical Engineering, Stanford University, 1979.

[34] Olavsrud, T., "HP Issues Battle Cry in High-End Unix Server Market," *ServerWatch, http://www.serverwatch.com/news/article.php/ 1399451*, 2000.

[35] Peir, J.-K., Lee, Y., and Hsu, W. W., "Capturing dynamic memory reference behavior with adaptive cache technology," in *the 8th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[36] Qureshi, M. K., Lynch, D. N., Mutlu, O., and Patt, Y. N., "A case for mlp-aware cache replacement," in *the 33rd International Symposium on Computer Architecture*, pp. 167–177, June 2006.

[37] Qureshi, M. K., Thompson, D., and Patt, Y. N., "The v-way cache: Demand-based associativity via global replacement," in *the 32nd International Symposium on Computer Architecture*, pp. 544–555, June 2005.

[38] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J., "Adaptive insertion policies for high-performance caching," in *The 34th International Symposium on Computer Architecture*, pp. 381–391, June 2007.

[39] Qureshi, M. K., Suleman, M. A., and Patt, Y. N., "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *the 13th International Symposium on High Performance Computer Architecture*, pp. 250–259, February 2007.

[40] Riley, N. and Zilles, C., "Probabilistic counter updates for predictor hysteresis and stratification," in *the 12th International Symposium on High Performance Computer Architecture*, pp. 110–120, February 2006.

[41] Rogers, B., Solihin, Y., and Prvulovic, M., "Efficient data protection for distributed shared memory multiprocessors," in *International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[42] Seznec, A., "A case for two-way skewed-associative caches," in *the 20th International Symposium on Computer Architecture*, pp. 169–178, May 1993.

[43] Shi, W., Lee, H.-H., Ghosh, M., and Lu, C., "Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 123–134, September 2004.

[44] Shi, W., Lee, H.-H., Ghosh, M., Lu, C., and Boldyreva, A., "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation," in *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[45] Shi, W., Lee, H.-H., Lu, C., and Ghosh, M., "Towards the Issues in Architectural Support for Protection of Software Execution," in *Proceedings of the Workshop on Architectureal Support for Security and Anti-virus*, pp. 1–10, October 2004.

[46] Silicon Graphics, Inc., "SGI Altix 3000 Data Sheet," *http://www.sgi.com/products/servers/altix*, 2004.

[47] Standard Performance Evaluation Corporation, "http://www.spec.org," 2004.

[48] Standard Performance Evaluation Corporation, "http://www.spec.org/cpu2006," 2006.

[49] Subramanian, R., Smaragdakis, Y., and Loh, G. H., "Adaptive cache: Effective shaping of cache behavior to workloads," in *the 39th International Symposium on Microarchitecture*, pp. 385–396, December 2006.

[50] Suh, G., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S., "Efficient Memory Integrity Verification and Encryption for Secure Processor," in *Proc. of the 36th Annual International Symposium on Microarchitecture*, 2003.

[51] Tyson, G., Farrens, M., Matthews, J., and Pleszkun, A. R., "A modified approach to data cache management," in *the 28th International Symposium on Microarchitecture*, pp. 93–103, 1995.

[52] Woo, S., Ohara, M., Torrie, E., Singh, J., and Gupta, A., "The splash-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 24–36, 1995.

[53] Yan, C., Rogers, B., Englender, D., Solihin, Y., and Prvulovic, M., "Improving cost, performance, and security of memory encryption and authentication," in *Proc. of the International Symposium on Computer Architecture*, 2006.

[54] Yang, B., Mishra, S., and Karri, R., "A high speed architecture for galois/counter mode of operation (gcm)," in *Cryptology ePrint Archive: Report 2005/146*, 2005.

[55] Yang, J., Zhang, Y., and Gao, L., "Fast Secure Processor for Inhibiting Software Piracy and Tampering," in *Proc. of the 36th Annual International Symposium on Microarchitecture*, 2003.

[56] Zhang, Y., Gao, L., Yang, J., Zhang, X., and Gupta, R., "SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors," in *International Symposium on High-Performance Computer Architecture*, February 2005.